Decorators

Introduction

With the introduction of Classes in TypeScript and ES6, there now exist certain scenarios that require additional features to support annotating or modifying classes and class members. Decorators provide a way to add both annotations and a meta-programming syntax for class declarations and members. Decorators are a stage 2 proposal for JavaScript and are available as an experimental feature of TypeScript.

> NOTE Decorators are an experimental feature that may change in future releases.

To enable experimental support for decorators, you must enable the `experimentalDecorators` compiler option either on the command line or in your `tsconfig.json`:

Command Line:

```
tsc --target ES5 --experimentalDecorators
```

tsconfig.json:

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

Decorators

A *Decorator* is a special kind of declaration that can be attached to a class declaration, method, accessor, property, or parameter. Decorators use the form `@expression`, where `expression` must evaluate to a function that will be called at runtime with information about the decorated declaration.

For example, given the decorator `@sealed` we might write the `sealed` function as follows:

```
function sealed(target) {
  // do something with 'target' ...
}
```

> NOTE You can see a more detailed example of a decorator in Class Decorators, below.

Decorator Factories

If we want to customize how a decorator is applied to a declaration, we can write a decorator factory. A *Decorator Factory* is simply a function that returns the expression that will be called by the decorator at runtime.

We can write a decorator factory in the following fashion:

```
function color(value: string) {
  // this is the decorator factory
  return function (target) {
    // this is the decorator
    // do something with 'target' and 'value'...
  };
}
```

> NOTE You can see a more detailed example of a decorator factory in Method Decorators, below.

Decorator Composition

Multiple decorators can be applied to a declaration, as in the following examples:

- On a single line:

  ```
  @f @g x
  ```

- On multiple lines:

```
@f
@g
x
```

When multiple decorators apply to a single declaration, their evaluation is similar to function composition in mathematics. In this model, when composing functions f and g, the resulting composite $(f \circ g)(x)$ is equivalent to $f(g(x))$.

As such, the following steps are performed when evaluating multiple decorators on a single declaration in TypeScript:

1. The expressions for each decorator are evaluated top-to-bottom.
2. The results are then called as functions from bottom-to-top.

If we were to use decorator factories, we can observe this evaluation order with the following example:

```
function f() {
  console.log("f(): evaluated");
  return function (
    target,
    propertyKey: string,
    descriptor: PropertyDescriptor
  ) {
    console.log("f(): called");
  };
}

function g() {
  console.log("g(): evaluated");
  return function (
    target,
    propertyKey: string,
    descriptor: PropertyDescriptor
  ) {
    console.log("g(): called");
  };
}

class C {
  @f()
  @g()
```

```
  method() {}
}
```

Which would print this output to the console:

```
f(): evaluated
g(): evaluated
g(): called
f(): called
```

Decorator Evaluation

There is a well defined order to how decorators applied to various declarations inside of a class are applied:

1. *Parameter Decorators*, followed by *Method*, *Accessor*, or *Property Decorators* are applied for each instance member.
2. *Parameter Decorators*, followed by *Method*, *Accessor*, or *Property Decorators* are applied for each static member.
3. *Parameter Decorators* are applied for the constructor.
4. *Class Decorators* are applied for the class.

Class Decorators

A *Class Decorator* is declared just before a class declaration. The class decorator is applied to the constructor of the class and can be used to observe, modify, or replace a class definition. A class decorator cannot be used in a declaration file, or in any other ambient context (such as on a `declare` class).

The expression for the class decorator will be called as a function at runtime, with the constructor of the decorated class as its only argument.

If the class decorator returns a value, it will replace the class declaration with the provided constructor function.

> NOTE Should you choose to return a new constructor function, you must take care to maintain the original prototype. The logic that applies decorators at runtime will **not** do this for you.

The following is an example of a class decorator (`@sealed`) applied to the `Greeter` class:

```
@sealed
class Greeter {
  greeting: string;
  constructor(message: string) {
```

```
    this.greeting = message;
  }
  greet() {
    return "Hello, " + this.greeting;
  }
}
```

We can define the `@sealed` decorator using the following function declaration:

```
function sealed(constructor: Function) {
  Object.seal(constructor);
  Object.seal(constructor.prototype);
}
```

When `@sealed` is executed, it will seal both the constructor and its prototype.

Next we have an example of how to override the constructor.

```
function classDecorator<T extends { new (...args: any[]): {} }>(
  constructor: T
) {
  return class extends constructor {
    newProperty = "new property";
    hello = "override";
  };
}

@classDecorator
class Greeter {
  property = "property";
  hello: string;
  constructor(m: string) {
    this.hello = m;
  }
}

console.log(new Greeter("world"));
```

Method Decorators

A *Method Decorator* is declared just before a method declaration. The decorator is applied to the *Property Descriptor* for the method, and can be used to observe, modify, or replace a method definition. A method decorator cannot be used in a declaration file, on an overload, or in any other ambient context (such as in a `declare` class).

The expression for the method decorator will be called as a function at runtime, with the following three arguments:

1. Either the constructor function of the class for a static member, or the prototype of the class for an instance member.
2. The name of the member.
3. The *Property Descriptor* for the member.

> NOTE The *Property Descriptor* will be `undefined` if your script target is less than `ES5`.

If the method decorator returns a value, it will be used as the *Property Descriptor* for the method.

> NOTE The return value is ignored if your script target is less than `ES5`.

The following is an example of a method decorator (`@enumerable`) applied to a method on the `Greeter` class:

```
class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }

  @enumerable(false)
  greet() {
    return "Hello, " + this.greeting;
  }
}
```

We can define the `@enumerable` decorator using the following function declaration:

```
function enumerable(value: boolean) {
  return function (
    target: any,
    propertyKey: string,
    descriptor: PropertyDescriptor
  ) {
    descriptor.enumerable = value;
```

```
  };
}
```

The `@enumerable(false)` decorator here is a decorator factory. When the `@enumerable(false)` decorator is called, it modifies the `enumerable` property of the property descriptor.

Accessor Decorators

An *Accessor Decorator* is declared just before an accessor declaration. The accessor decorator is applied to the *Property Descriptor* for the accessor and can be used to observe, modify, or replace an accessor's definitions. An accessor decorator cannot be used in a declaration file, or in any other ambient context (such as in a `declare` class).

> NOTE TypeScript disallows decorating both the `get` and `set` accessor for a single member. Instead, all decorators for the member must be applied to the first accessor specified in document order. This is because decorators apply to a *Property Descriptor*, which combines both the `get` and `set` accessor, not each declaration separately.

The expression for the accessor decorator will be called as a function at runtime, with the following three arguments:

1. Either the constructor function of the class for a static member, or the prototype of the class for an instance member.
2. The name of the member.
3. The *Property Descriptor* for the member.

> NOTE The *Property Descriptor* will be `undefined` if your script target is less than `ES5`.

If the accessor decorator returns a value, it will be used as the *Property Descriptor* for the member.

> NOTE The return value is ignored if your script target is less than `ES5`.

The following is an example of an accessor decorator (`@configurable`) applied to a member of the `Point` class:

```
class Point {
  private _x: number;
  private _y: number;
  constructor(x: number, y: number) {
    this._x = x;
    this._y = y;
  }

  @configurable(false)
  get x() {
    return this._x;
```

```
  }

  @configurable(false)
  get y() {
    return this._y;
  }
}
```

We can define the `@configurable` decorator using the following function declaration:

```
function configurable(value: boolean) {
  return function (
    target: any,
    propertyKey: string,
    descriptor: PropertyDescriptor
  ) {
    descriptor.configurable = value;
  };
}
```

Property Decorators

A *Property Decorator* is declared just before a property declaration. A property decorator cannot be used in a declaration file, or in any other ambient context (such as in a `declare` class).

The expression for the property decorator will be called as a function at runtime, with the following two arguments:

1. Either the constructor function of the class for a static member, or the prototype of the class for an instance member.
2. The name of the member.

> NOTE A *Property Descriptor* is not provided as an argument to a property decorator due to how property decorators are initialized in TypeScript. This is because there is currently no mechanism to describe an instance property when defining members of a prototype, and no way to observe or modify the initializer for a property. The return value is ignored too. As such, a property decorator can only be used to observe that a property of a specific name has been declared for a class.

We can use this information to record metadata about the property, as in the following example:

```
class Greeter {
  @format("Hello, %s")
  greeting: string;
```

```
  constructor(message: string) {
    this.greeting = message;
  }
  greet() {
    let formatString = getFormat(this, "greeting");
    return formatString.replace("%s", this.greeting);
  }
}
```

We can then define the `@format` decorator and `getFormat` functions using the following function declarations:

```
import "reflect-metadata";

const formatMetadataKey = Symbol("format");

function format(formatString: string) {
  return Reflect.metadata(formatMetadataKey, formatString);
}

function getFormat(target: any, propertyKey: string) {
  return Reflect.getMetadata(formatMetadataKey, target, propertyKey);
}
```

The `@format("Hello, %s")` decorator here is a decorator factory. When `@format("Hello, %s")` is called, it adds a metadata entry for the property using the `Reflect.metadata` function from the `reflect-metadata` library. When `getFormat` is called, it reads the metadata value for the format.

> NOTE This example requires the `reflect-metadata` library. See Metadata for more information about the `reflect-metadata` library.

Parameter Decorators

A *Parameter Decorator* is declared just before a parameter declaration. The parameter decorator is applied to the function for a class constructor or method declaration. A parameter decorator cannot be used in a declaration file, an overload, or in any other ambient context (such as in a `declare` class).

The expression for the parameter decorator will be called as a function at runtime, with the following three arguments:

1. Either the constructor function of the class for a static member, or the prototype of the class for an instance member.
2. The name of the member.
3. The ordinal index of the parameter in the function's parameter list.

> NOTE A parameter decorator can only be used to observe that a parameter has been declared on a method.

The return value of the parameter decorator is ignored.

The following is an example of a parameter decorator (`@required`) applied to parameter of a member of the `Greeter` class:

```
class Greeter {
  greeting: string;

  constructor(message: string) {
    this.greeting = message;
  }

  @validate
  greet(@required name: string) {
    return "Hello " + name + ", " + this.greeting;
  }
}
```

We can then define the `@required` and `@validate` decorators using the following function declarations:

```
import "reflect-metadata";

const requiredMetadataKey = Symbol("required");

function required(
  target: Object,
  propertyKey: string | symbol,
  parameterIndex: number
) {
  let existingRequiredParameters: number[] =
    Reflect.getOwnMetadata(requiredMetadataKey, target, propertyKey) || [];
  existingRequiredParameters.push(parameterIndex);
  Reflect.defineMetadata(
    requiredMetadataKey,
    existingRequiredParameters,
    target,
    propertyKey
```

```
  );
}

function validate(
  target: any,
  propertyName: string,
  descriptor: TypedPropertyDescriptor<Function>
) {
  let method = descriptor.value;
  descriptor.value = function () {
    let requiredParameters: number[] = Reflect.getOwnMetadata(
      requiredMetadataKey,
      target,
      propertyName
    );
    if (requiredParameters) {
      for (let parameterIndex of requiredParameters) {
        if (
          parameterIndex >= arguments.length ||
          arguments[parameterIndex] === undefined
        ) {
          throw new Error("Missing required argument.");
        }
      }
    }

    return method.apply(this, arguments);
  };
}
```

The `@required` decorator adds a metadata entry that marks the parameter as required. The `@validate` decorator then wraps the existing `greet` method in a function that validates the arguments before invoking the original method.

> NOTE This example requires the `reflect-metadata` library. See Metadata for more information about the `reflect-metadata` library.

Metadata

Some examples use the `reflect-metadata` library which adds a polyfill for an experimental metadata API. This library is not yet part of the ECMAScript (JavaScript) standard. However, once decorators are officially adopted as part of the ECMAScript standard these extensions will be proposed for adoption.

You can install this library via npm:

```
npm i reflect-metadata --save
```

TypeScript includes experimental support for emitting certain types of metadata for declarations that have decorators. To enable this experimental support, you must set the `emitDecoratorMetadata` compiler option either on the command line or in your `tsconfig.json`:

Command Line:

```
tsc --target ES5 --experimentalDecorators --emitDecoratorMetadata
```

tsconfig.json:

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

When enabled, as long as the `reflect-metadata` library has been imported, additional design-time type information will be exposed at runtime.

We can see this in action in the following example:

```
import "reflect-metadata";

class Point {
  x: number;
  y: number;
}

class Line {
  private _p0: Point;
  private _p1: Point;
```

```
  @validate
  set p0(value: Point) {
    this._p0 = value;
  }
  get p0() {
    return this._p0;
  }

  @validate
  set p1(value: Point) {
    this._p1 = value;
  }
  get p1() {
    return this._p1;
  }
}

function validate<T>(
  target: any,
  propertyKey: string,
  descriptor: TypedPropertyDescriptor<T>
) {
  let set = descriptor.set;
  descriptor.set = function (value: T) {
    let type = Reflect.getMetadata("design:type", target, propertyKey);
    if (!(value instanceof type)) {
      throw new TypeError("Invalid type.");
    }
    set.call(target, value);
  };
}
```

The TypeScript compiler will inject design-time type information using the `@Reflect.metadata` decorator. You could consider it the equivalent of the following TypeScript:

```
class Line {
  private _p0: Point;
  private _p1: Point;

  @validate
  @Reflect.metadata("design:type", Point)
  set p0(value: Point) {
```

```
    this._p0 = value;
  }
  get p0() {
    return this._p0;
  }

  @validate
  @Reflect.metadata("design:type", Point)
  set p1(value: Point) {
    this._p1 = value;
  }
  get p1() {
    return this._p1;
  }
}
```

NOTE Decorator metadata is an experimental feature and may introduce breaking changes in future releases.

The TypeScript docs are an open source project. Help us improve these pages by sending a Pull Request ❤

Contributors to this page:

 OT MH  RC 15+

Last updated: Feb 08, 2021

This page loaded in 1.548 seconds.

TypeScript

Made with ♥ in Redmond, Boston, SF & Dublin

Microsoft

Privacy

Using TypeScript

- Get Started
- Download
- Community
- Playground
- TSConfig Ref
- Why TypeScript
- Design
- ▼ Code Samples

Community

- Get Help
- Blog
- GitHub Repo
- Community Chat
- @TypeScript
- Stack Overflow
- Web Updates
- Web Repo

/