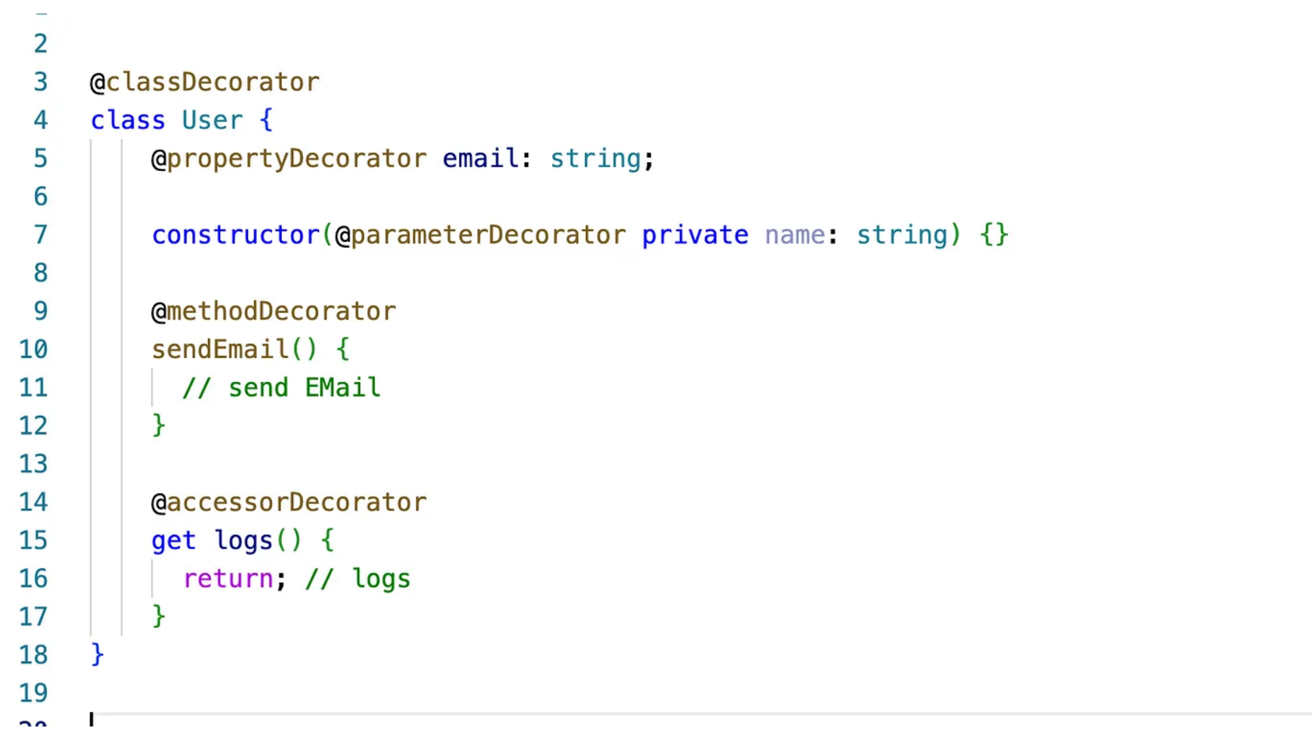
Get unlimited access to the best of Medium for less than $1/week. Become a member

What are “decorators” in typescript and how to use “decorators”?

Hemant · Follow
11 min read · Sep 3

Listen Share More



```

@classDecorator
class User {
    @propertyDecorator email: string;

    constructor(@parameterDecorator private name: string) {}

    @methodDecorator
    sendEmail() {
      // send EMail
    }

    @accessorDecorator
    get logs() {
      return; // logs
    }
}

```

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.

Stage 3 decorator support has been available since Typescript 5.0

How to use decorators:

To enable experimental support for decorators, you must enable the `experimentalDecorators` compiler option either on the command line using `tsc --target ES5 --experimentalDecorators` or in your `tsconfig.json`:

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

Consider a user class with a method `greet`

```
class User {
  constructor(private name: string, private age: number) {}

  greet() {
    console.log(`Hello, my name is ${this.name}.`);
  }

  printAge() {
    console.log(`I am ${this.age} years old`);
  }
}

const user = new User("Ron", 25);
user.greet();
user.printAge();



Output:
Hello, my name is Ron.
I am 25 years old
```

Now we want to log when each function execution starts and ends:

```
class User {
  constructor(private name: string, private age: number) {}
```

```
    greet() {
      console.log('start: greet')
      console.log(`Hello, my name is ${this.name}.`);
      console.log('end: greet')
    }

    printAge() {
      console.log('start: printAge')
      console.log(`I am ${this.age} years old`);
      console.log('end: printAge')
    }
  }

  const user = new User("Ron", 25);
  user.greet();
  user.printAge();



  Output:
  start: greet
  Hello, my name is Ron.
  end: greet
  start: printAge
  I am 25 years old
  end: printAge
```

> **Now think about it, Can we reuse this logic of logging function execution start and end? Yes, we can create a wrapper function that logs the start and end of the wrapped function. That's what decorators do for us in a neat and clean way. Let's see how**

It's super easy to create a decorator: Just create a function called `logger` :

```
function logger(originalMethod: any, _context: any) {
  function replacementMethod(this: any, ...args: any[]) {
    console.log("start:", originalMethod.name);
    const result = originalMethod.call(this, ...args);
    console.log("end:", originalMethod.name);
    return result;
  }

  return replacementMethod;
}
```

That's it. We are ready to decorate the methods. Let's use decorators in the above example:

```
class User {
  constructor(private name: string, private age: number) {}

  @logger
  greet() {
    console.log(`Hello, my name is ${this.name}.`);
  }

  @logger
  printAge() {
    console.log(`I am ${this.age} years old`);
  }
}

const user = new User("Ron", 25);
user.greet();
user.printAge();



Output:
start: greet
Hello, my name is Ron.
end: greet
start: printAge
I am 25 years old
end: printAge
```

Easy. Is't it?

Typescript supports multiple decorators:

```
  @logger
  @xyz
  greet() {
    console.log(`Hello, my name is ${this.name}.`);
  }
```

Let's see the order of execution when multiple decorators are applied.

```
class User {
  constructor(private name: string, private age: number) {}

  @logger2
  @logger1
  greet() {
    console.log(`Hello, my name is ${this.name}.`);
  }
}

const user = new User("Ron", 25);
user.greet();


function logger1(originalMethod: any, _context: any) {
  function replacementMethod(this: any, ...args: any[]) {
    console.log("log1");
    const result = originalMethod.call(this, ...args);
    return result;
  }

  return replacementMethod;
}

function logger2(originalMethod: any, _context: any) {
  function replacementMethod(this: any, ...args: any[]) {
    console.log("log2");
    const result = originalMethod.call(this, ...args);
    return result;
  }

  return replacementMethod;
}





Output:
log2
log1
Hello, my name is Ron.
```

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:**
>
> **The expressions for each decorator are evaluated top-to-bottom.**
>
> **The results are then called as functions from bottom-to-top.**

Example of Well-Typed Decorator:

```
function loggedMethod<This, Args extends any[], Return>(
    target: (this: This, ...args: Args) => Return,
    context: ClassMethodDecoratorContext<This, (this: This, ...args: Args) => R
) {
    const methodName = String(context.name);

    function replacementMethod(this: This, ...args: Args): Return {
        console.log(`LOG: Entering method '${methodName}'.`)
        const result = target.call(this, ...args);
        console.log(`LOG: Exiting method '${methodName}'.`)
        return result;
    }

    return replacementMethod;
}
```

Types of decorators:

1. 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. “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.”

Here is an example where we will try to set created property using class decorator.

```
class User {
  [x: string]: any;
  constructor(public name: string) {}
}

const user = new User('John')
console.log(user.name, user.created)

// Output:
John undefined
```

With class decorator

```
@BaseEntity
class User {
  [x: string]: any;
  constructor(public name: string) {}
}

function BaseEntity(ctr: Function) {
  ctr.prototype.created = new Date().toISOString();
}

const user = new User('John')
console.log(user.name, user.created)
```

2. 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). We have already seen the example of method decorators so won't be going into any further details:

```
class User {
  constructor(private name: string, private age: number) {}

  @logger
  greet() {
    console.log(`Hello, my name is ${this.name}.`);
  }

  @logger
  printAge() {
    console.log(`I am ${this.age} years old`);
  }
}

const user = new User("Ron", 25);
user.greet();
user.printAge();

Output:
start: greet
Hello, my name is Ron.
end: greet
start: printAge
I am 25 years old
end: printAge
```

3. 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;
  constructor(x: number, y: number) {
    this._x = x;
  }

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


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

4. 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.

Here’s an example of a property decorator in TypeScript that validates if a property’s value is a valid email address:

```
// Property Decorator for Email Validation
function ValidateEmail(target: any, propertyKey: string) {
  const privateFieldName = `_${propertyKey}`;

  // Store the original setter method
  const originalSetter = Object.getOwnPropertyDescriptor(target, propertyKey)?.

  // Define a new setter for the property
  const newSetter = function (value: any) {
    if (!isValidEmail(value)) {
      throw new Error(`Invalid email address for property "${propertyKey}".`);
    }
    this[privateFieldName] = value;
  };

  // Replace the property's setter method
  Object.defineProperty(target, propertyKey, {
    set: newSetter,
    get() {
      return this[privateFieldName];
    },
    enumerable: true,
    configurable: true,
  });
}

// Helper function to validate email addresses
function isValidEmail(email: string): boolean {
  // Regular expression for a simple email validation
  const emailPattern = /^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/;
  return emailPattern.test(email);
}

class User {
  @ValidateEmail
  email: string = 'test@example.com';

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

const user = new User('john@example.com');

console.log(user.email); // john@example.com

try {
  user.email = 'invalid-email'; // This will throw an error
} catch (error) {
  console.error(error.message); // Invalid email address for property "email".
}




// Output:
john@example.com
Invalid email address for property "email".
```

- We define a property decorator `ValidateEmail` that checks if the assigned value to the property is a valid email address using the `isValidEmail` helper function.
- The `newSetter` function checks if the provided value is a valid email address. If not, it throws an error.
- We apply the `@ValidateEmail` decorator to the `email` property of the `User` class.
- When we create an instance of `User`, we set the `email` property to a valid email address, which works as expected.
- If you attempt to set the `email` property to an invalid email address (e.g., `'invalid-email'`), the decorator will throw an error indicating that the email address is invalid.

5. 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.

Here's an example of a parameter decorator that validates if a method parameter is a valid email address using a simple regular expression:

```
// Parameter Decorator for Email Validation
function ValidateEmail(target: any, methodName: string, parameterIndex: number)
  const originalMethod = target[methodName];

  target[methodName] = function (...args: any[]) {
    const paramValue = args[parameterIndex];

    // Regular expression for a simple email validation
    const emailPattern = /^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/;

    if (!emailPattern.test(paramValue)) {
      throw new Error(`Invalid email address provided for parameter at index ${
    }

    return originalMethod.apply(this, args);
  };
}

class ExampleClass {
  // Apply the parameter decorator to validate email parameter
  sendEmail(@ValidateEmail email: string) {
    console.log(`Sending email to ${email}`);
  }
}

const exampleInstance = new ExampleClass();

// This will work
exampleInstance.sendEmail("example@email.com");

// This will throw an error due to email validation
try {
  exampleInstance.sendEmail("invalid-email");
} catch (error) {
  console.error(error.message); // Invalid email address provided for parameter
}
```

1. We define a parameter decorator called `ValidateEmail` that checks if the provided parameter is a valid email address using a regular expression.
2. We create an `ExampleClass` class with a `sendEmail` method. We apply the `@ValidateEmail` decorator to validate the `email` parameter.
3. When we call the `sendEmail` method, it checks whether the provided email parameter matches the email validation regular expression. If it doesn't match, it throws an error.
4. We demonstrate two calls to the `sendEmail` method—one with a valid email and another with an invalid email that results in an email validation error.

This example shows how you can use a parameter decorator to perform simple email validation for method parameters. Depending on your specific requirements, you can adjust the regular expression or add more complex email validation logic as needed.

Decorators in TypeScript provide a powerful mechanism for modifying or adding behavior to various parts of your code. Here are some common use cases for decorators:

1. **Logging and Debugging:** You can use decorators to log method calls, function parameters, or property access to aid in debugging and tracing the flow of your application.
2. **Validation:** Decorators can be used for input validation, ensuring that function parameters or property values meet specific criteria or constraints.
3. **Memoization:** You can implement memoization by using decorators to cache function results based on their input parameters, improving performance for expensive calculations.
4. **Authentication and Authorization:** Decorators can check user authentication or authorization before allowing access to certain methods or routes in web applications.

5. **Dependency Injection:** In frameworks like Angular, decorators are used for dependency injection, allowing you to specify which services should be injected into your classes or components.
6. **Route Handling (Web Applications)**: In web frameworks like Express.js or Nest.js, decorators are used to define routes and request handlers for HTTP endpoints.
7. **Data Transformation:** You can use decorators to transform data before it's processed, such as converting JSON objects into class instances with custom logic.
8. **Caching**: Decorators can be used to cache data retrieval methods, reducing the load on external data sources.
9. **Timing and Profiling:** Decorators can measure the execution time of functions, helping with performance profiling and optimization.
10. **Logging Frameworks:** In custom logging frameworks, decorators can be applied to methods to log specific events or actions in your application.
11. **Validation Frameworks:** You can create custom validation decorators to ensure that data conforms to specific rules or constraints.
12. **Database Mapping:** In Object-Relational Mapping (ORM) libraries like TypeORM, decorators are used to map class properties to database columns.
13. **Property Access Control**: Decorators can enforce access control policies on class properties, ensuring that only authorized code can access or modify them.
14. **Singleton Pattern**: Decorators can be used to implement the Singleton design pattern, ensuring that only one instance of a class is created.
15. **Custom Middleware:** In web frameworks, decorators can be used to create custom middleware functions that execute before or after the main request handler.
16. **Internationalization and Localization:** Decorators can be applied to text properties or methods to handle language translation and localization.
17. **Error Handling:** You can use decorators to centralize error handling logic, making it easier to handle exceptions consistently across your codebase.

18. **Event Handling:** Decorators can be used to register event listeners and handlers for specific events in your application.

19. **Type Checking and Transformation:** Decorators can perform type checking and data transformation, ensuring that data conforms to expected types and formats.

20. **Custom Annotations:** You can create custom annotations or metadata to provide additional information about classes, methods, or properties that can be used by

your classes, methods, and properties, making your code more modular and maintainable.

Typescript Decorators JavaScript

Follow

Written by Hemant

19 Followers

Passionate Developer on a continuous quest for innovation. Let's explore the digital frontier together.

More from Hemant