Creating and Using Decorators in TypeScript 5

Introduction to Decorators

Ivan Mushketyk

@mushketyk | ivanm.io

Creating and Using Decorators in TypeScript 5

Version Check

Version Check

This course was created by using:

- TypeScript 5.0 or later
- Some content requires TypeScript 5.2

```
class WeatherAPI {
  @cached({ timeToLive: 1000 })
  getForecast(city: string): Promise<Forecast> {
    return {
      ...
    }
  }
}

function cached(cachingConfig: CachingConfig) {
  ...
}
```

What Are Decorators?

- A function executed in runtime
- Modifies a class or one of its members

Types of Decorators

Classes

Methods

Accessors

Properties

Auto-accessors

Metaprogramming

Metaprogramming is a programming technique in which computer programs have the ability to treat other programs as their data. It means that a program can be designed to read, generate, analyze or transform other programs, and even modify itself while running

Link: https://en.wikipedia.org/wiki/Metaprogramming

Nest.js Example

```
import { Controller, Inject, Get, Param } from '@nestjs/common';

@Controller('weather')
export class WeatherAPI {

  @Inject('FORECAST_CLIENT')
  private readonly forecastClient: ForecastClient

  @Get(':cityName')
  async weatherForcast(@Param() params): Promise<Forecast> {
    return this.forecastClient.getForecast(params.cityName);
  }
}
```

TypeORM Example

```
import { Entity, PrimaryGeneratedColumn, Column } from 'typeorm'

@Entity()
export class User {
  @PrimaryGeneratedColumn()
  id: number

  @Column()
  name: string

  @Column()
  isAdmin: boolean
}

const person = await dataSource.manager.findOneBy(User, { id: 1 })
```

Two Types of Decorators

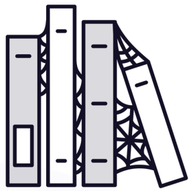
TypeScript 5 decorators	VS	Pre-TypeScript 5 Decorators
New decorators implementation		"Experimental decorators"
Implement a new JavaScript standard for decorators		Implement an older version of the decorators proposal
Enabled by default		Should be explicitly enabled
Only available in TypeScript 5 and later		Still supported
Can switch between new and old decorators		Widely used by popular libraries

Plan for This Course

Basics of decorators

Implement method decorators

Implement a custom framework using decorators



Review of the "experimental decorators"

Important Prerequisites

Will be writing a lot of functions

- Might get confusing

Learn/refresh important prerequisites

- Higher-order functions
- Currying
- Closure

Higher-order Functions

Using functions as parameters

```
function process(input: number[], func): number[] {
  const result = []
  for (let i = 0; i < input.length; i++) {
    const processed = func(input[i])
    result.push(processed)
  }
  return result
}
```

Higher-order Functions

Using functions as parameters

```
function process(input: number[], func: (n: number) => number): number[] {
  const result = []
  for (let i = 0; i < input.length; i++) {
    const processed = func(input[i])
    result.push(processed)
  }
  return result
}
```

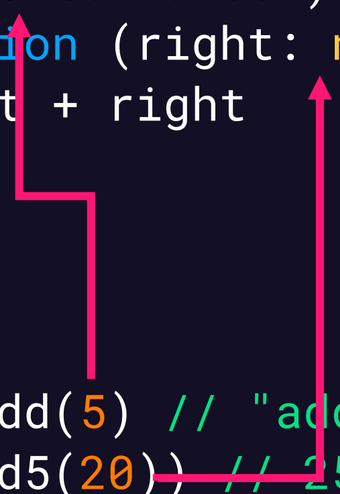
```ts
function process(input: number[], func: (n: number) => number): number[] {
  const result = []
  for (let i = 0; i < input.length; i++) {
    const processed = func(input[i])
    result.push(processed)
  }
  return result
}

const arr = [1, 2, 3]
const result = process(arr, (n) => n * 2)
console.log(result)
```

```
> [2, 4, 6]
```

Currying

Create a function using another function

```
function add(left: number) {
  return function (right: number) {
    return left + right
  }
}


const add5 = add(5) // "add5" is a function
console.log(add5(20)) // 25
```



Currying

Create a function using another function

```
function add(left: number) {
  return function (right: number) {
    return left + right
  }
}


const add5 = add(5) // "add5" is a function
console.log(add5(20)) // 25
console.log(add5(30)) // 35
```

Closure

A function has access to its surrounding scope

```
function add(left: number) {
  return function (right: number) {
    return left + right
  }
}


const add5 = add(5)
console.log(add5(20)) // 25
console.log(add5(30)) // 35
```

Method Decorators

How method decorators work

How to implement a method decorator

- Examples of decorators
- Limitations

Will look at other decorator types later

What Is a Method Decorator?

```
function myDecorator(target, context) {
  return function (this, ...args) {
    return 'Hello world!'
  }
}
class GitHubClient {
  @myDecorator
  public async getRepos() {
    console.log('Getting GitHub repos...')
    ...
  }
}
```

```
function myDecorator(target, context) {
  return function (this, ...args) {
    return 'Hello world!'
  }
}
class GitHubClient {
  @myDecorator
  public async getRepos() {
    console.log('Getting GitHub repos...')
    ...
  }
}
console.log(new GitHubClient().getRepos())
```

```
> Hello world!
```

Logging Decorator

```
// Write a log message before and after a target method's execution

function log(target, context) {
  return function (this, ...args) {
    console.log(`Calling method at ${new Date().toISOString()}`)
    const result = target.apply(this, args)
    console.log(`Method finished at ${new Date().toISOString()}`)
    return result
  }
}
```

```
function log(target, context) {
  return function (this, ...args) {
    console.log(`Calling method at ${new Date().toISOString()}`)
    const result = target.apply(this, args)
    console.log(`Method finished at ${new Date().toISOString()}`)
    return result
  }
}
class GitHubClient {
  @log
  public getRepos() {
    console.log('Getting GitHub repos...')
  }
}
new GitHubClient().getRepos())
```

```
function log(target, context) {
  return function (this, ...args) {
    console.log(`Calling method at ${new Date().toISOString()}`)
    const result = target.apply(this, args)
    console.log(`Method finished at ${new Date().toISOString()}`)
    return result
  }
}
class GitHubClient {
  @log
  public getRepos() {
    console.log('Getting GitHub repos...')
  }
}
new GitHubClient().getRepos())
```

```
> Calling method at 2023-09-10T16:54:09.693Z
> Getting GitHub repos...
> Method finished at 2023-09-10T16:54:09.793Z

> Calling method at 2023-09-10T16:54:10.683Z
> Getting GitHub repos...
> Method finished at 2023-09-10T16:54:10.792Z
```

Decorator Definition

```
type ClassMethodDecorator = (
  value: Function,
  context: {
    kind: 'method'
    name: string | symbol
    static: boolean
    private: boolean
    access: { get: () => unknown }
    addInitializer(initializer: () => void): void
    metadata: Record<PropertyKey, unknown>
  },
) => Function | void
```

Getting a Method Name

```
// Write a log message before and after a target method execution
function log(target, context) {
  return function (this, ...args) {
    console.log(`Calling method at ${new Date().toISOString()}`)
    const result = target.apply(this, args)
    console.log(`Method finished at ${new Date().toISOString()}`)
    return result
  }
}
```

Getting a Method Name

```
// Write a log message before and after a target method execution
function log(target, context) {
  return function (this, ...args) {
    console.log(`Calling method ${context.name} at ${new Date().toISOString()}`)
    const result = target.apply(this, args)
    console.log(`Method finished ${context.name} at ${new Date().toISOString()}`)
    return result
  }
}
```

```
function log(target, context) {
  return function (this, ...args) {
    console.log(`Calling method ${context.name} at ${new Date().toISOString()}`)
    const result = target.apply(this, args)
    console.log(`Method finished ${context.name} at ${new Date().toISOString()}`)
    return result
  }
}
class GitHubClient {
  @log
  public getRepos() {
    console.log('Getting GitHub repos...')
  }
}
```

```
> Calling method getRepos at 2023-09-10T16:54:09.693Z
> Getting Getting GitHub repos...
> Method finished getRepos at 2023-09-10T16:54:09.793Z
```

Some Applications of Method Decorators

Logging

Metrics/Tracing

Caching

Transactions

Callbacks

Method Decorators Limitations

Regular functions

There is a proposal to add support for regular functions

Constructors

Will learn about a workaround for this later

Demo

Implement our first method decorator

@retry decorator

- Retry a failed method automatically
- Improves application reliability

Simplified implementation

Configuring Decorators

Our current decorator is limiting

Users would like to change retry parameters

- How long to wait
- How many tries to make
- etc.

Decorator factory

- A function that create decorators

Decorator Function

```
// Decorator function
function log(target: any, context: any) {
  // Decorated method
  const resultMethod = async function (this, ...args) {
    logger.log(`Running method ${context.name}`, Logger.INFO)

    const res = target.apply(this, args)

    logger.log(`Finished method ${context.name}`, Logger.INFO)
    return res
  }
  return resultMethod
}
```

Decorator Function

```
// Decorator function
function log(target: any, context: any) {
  // Decorated method
  const resultMethod = async function (this, ...args) {
    logger.log(`Running method ${context.name}`, Logger.INFO)

    const res = target.apply(this, args)

    logger.log(`Finished method ${context.name}`, Logger.INFO)
    return res
  }
  return resultMethod
}
```

Decorator Factory

```
// Decorator factory
function log(logLevel: LogLevel) {
  // Decorator function
  return function (target: any, context: any) {
    // Decorated method
    const resultMethod = async function (this, ...args) {
      logger.log(`Running method ${context.name}`, logLevel)

      const res = target.apply(this, args)

      logger.log(`Finished method ${context.name}`, logLevel)
      return res
    }
    return resultMethod
  }
}
```

Using a Decorator Factory

```
// Decorator factory
function log(logLevel: LogLevel) {
  // Decorator function
  return function (target: any, context: any) {
    // Decorated method
    ...
    return resultMethod
  }
}

class GitHubClient {

  @log(Logger.INFO)
  getRepoInfo(repo: string) { ... }
}
```

Using a Decorator Factory

```
function log(target: any, context: any) {
  // Decorated method
  const resultMethod = async function (this, ...args) {
    logger.log(`Running method ${context.name}`, Logger.INFO)
    const res = target.apply(this, args)
    logger.log(`Finished method ${context.name}`, Logger.INFO)
    return res
  }
  return resultMethod
}


class GitHubClient {
  // NOT ALLOWED
  @log()
  getRepoInfo(repo: string) {}
}
```

Using a Decorator Factory

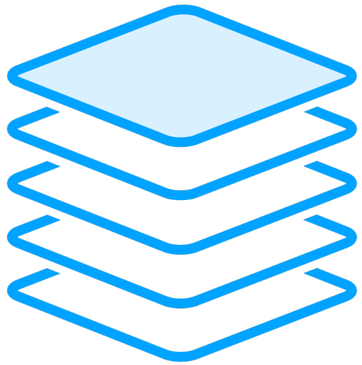
```
function log(target: any, context: any) {
  // Decorated method
  const resultMethod = async function (this, ...args) {
    logger.log(`Running method ${context.name}`, Logger.INFO)
    const res = target.apply(this, args)
    logger.log(`Finished method ${context.name}`, Logger.INFO)
    return res
  }
  return resultMethod
}


class GitHubClient {
  // ALLOWED
  @log
  getRepoInfo(repo: string) {}
}
```

Demo

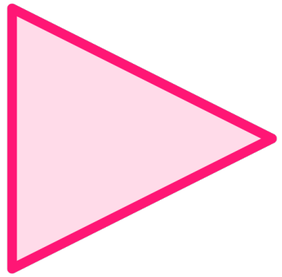
Implement a decorator factory for @retry

- Add two parameters
  - Number of retries
  - Delay between retries

Multiple Decorators



How to apply multiple decorators



How several decorators are executed

Implement a new decorator

Multiple Decorators

```
function time(metricName: string) {
  ...
}


function retry(delayMs: number, maxAttempts: number {
  ...
}
class GitHubClient {

  @time('get-report-info.duration')
  @retry(1000, 3)
  getRepoInfo(repo: string) {
    …
  }
}
```

Multiple Decorators

```
class GitHubClient {

  @time('get-report-info.duration')
  @retry(1000, 3)
  getRepoInfo(repo: string) {
    …
  }
}
```

Multiple Decorators

```
class GitHubClient {

  @time('get-report-info.duration')
  @retry(1000, 3)
  getRepoInfo(repo: string) {
    …
  }
}
```

Decorators Execution Order

github-client.ts

```
class GitHubClient {

  @time('get-report-info.duration')
  @retry(1000, 3)
  getRepoInfo(repo: string) {
    …
  }
}
```

```
> @time started
> @retry started
> getRepoInfo()
> @retry finished
> @time finished
```

```
class GitHubClient {

  @time('get-report-info.duration')
  @retry(1000, 3)
  getRepoInfo(repo: string) {...}
}
```

- Measure total duration
- Retry method up to three times

```
class GitHubClient {

  @retry(1000, 3)
  @time('get-report-info.duration')
  getRepoInfo(repo: string) {...}
}
```

- Retry decorated function call
- Measure individual calls

Demo

Implement @log decorator

- Write logs messages when a method is executed

Apply multiple decorators

Decorator Patterns

Class method as decorator

Decorators composition

Collecting data in decorators

Class Method as a Decorator

```
function retry(delayMs: number, maxAttempts: number) {
    ...
}
```

Class Method as a Decorator

```
class Retrier {
  constructor(private delayMs: number, private maxAttempts: number) {}

  // Returning a decorator
  retry() {
    // Decorator
    return function (target: any, context: any) {
      // New method that captures time metric
      const resultMethod = async function (this: any, ...args: any[]) {
        ...
      }
    }
  }
}
```

Class Method as a Decorator

```
class Retrier {
  constructor(private delayMs: number, private maxAttempts: number) {}

  // Returning a decorator
  retry() {
    // Decorator
    return function (target: any, context: any) {
      // New method that captures time metric
      const resultMethod = async function (this: any, ...args: any[]) {
        ...
      }
    }
  }
}
```

Class Method as a Decorator

```
class Retrier {
  constructor(private delayMs: number, private maxAttempts: number) {}

  // Returning a decorator
  retry() {
    // Decorator
    return function (target: any, context: any) {...}
  }
}

const retrier = new Retrier(1000, 3)

class GitHubClient {

  @retrier.retry()
  getRepoInfo(repo: string) {...}
}
```

Decorators Composition

```
function time(target: any, context: any) {...}
function cache(target: any, context: any) {...}
function retry(target: any, context: any) {...}


class WeatherAPI {


  @time
  @cache
  @retry
  getWeatherForecast() {
    …
  }
}
```

Decorators Composition

```
function time(target: any, context: any) {...}
function cache(target: any, context: any) {...}
function retry(target: any, context: any) {...}


class WeatherAPI {




  @safeCaching
  getWeatherForecast() {
    …
  }
}
```

Decorators Composition

```
function time(target: any, context: any) {...}
function cache(target: any, context: any) {...}
function retry(target: any, context: any) {...}


function safeCaching(target: any, context: any){
  const withRetry = retry(target, context)
  const withCache = cache(withRetry, context)
  const withTime = time(withCache, context)

  return withTime
}
```

Decorators Composition

```
function safeCaching(target: any, context: any){
  const withRetry = retry(target, context)
  const withCache = cache(withRetry, context)
  const withTime = time(withCache, context)

  return withTime
}

class WeatherAPI {


  @safeCaching
  getWeatherForecast() {
    ...
  }
}
```

Decorators Composition

```
// Decorator factory
function retry(retryOptions: RetryOptions) {
  // Decorator function
  return function (target: any, context: any) {
    ...
  }
}



function safeCaching(target: any, context: any){
  const withRetry = retry(target, context)
  const withCache = cache(withRetry, context)
  const withTime = time(withCache, context)

  return withTime
}
```

Decorators Composition

```
// Decorator factory
function retry(retryOptions: RetryOptions) {
  // Decorator function
  return function (target: any, context: any) {
    ...
  }
}



function safeRetry(target: any, context: any){
  const retryDecorator = retry({maxRetryAttempts: 4, delayMs: 500})
  const withRetry = retryDecorator(target, context)
  const withCache = cache(withRetry, context)
  const withTime = time(withCache, context)

  return withTime
}
```

Collecting Data in Decorators

Need to collect data from every application

Look at different ways of how to implement this

- From: "JavaScript metaprogramming with the 2022-03 decorators API" by Dr. Axel Rauschmayer

Collecting Metrics

```
function time(metricName: string) {
  return function (target: any, context: any) {...}
}

class GitHubClient {
  @time('fetch-user-details.duration')
  async fetchUserDetails() {...}

  @time('fetch-user-repositories.duration')
  async fetchUserRepositories() {...}
}

for (const metric of allMetrics) {
  reportValue(metric)
}
```

Using a Separate Variable

```
const metricValues = new Map<string, Array<number>>()


function time(metricName: string) {
  metricValues.set(metricName, [])
  return function (target: any, context: any) {
    ...
  }
}
```

Using a Class

```
class Metrics {
  private metricValues = new Map<string, Array<number>>()

  time(metricName: string) {
    this.metricValues.set(metricName, [])
    return function (target: any, context: any) {
      ...
    }
  }
}
```

Factory Function

```
function metricsFactory() {
  const metricValues = new Map<string, Array<number>>()

  function time(metricName: string) {
    metricValues.set(metricName, [])
    return function (target: any, context: any) {
      ...
    }
  }


  return {metricValues, time}
}


const {metricValues, time} = metricsFactory()
```

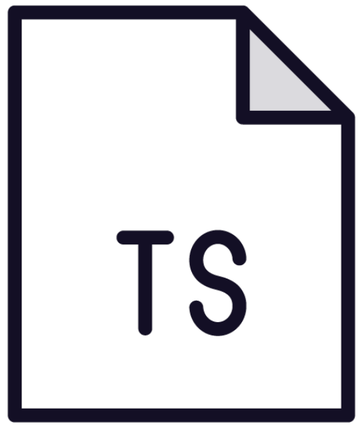
Demo

Class method as a decorator

- Measure time it took a method to execute

Decorators composition

Defining Typed Decorators



So far were mostly using "any" types

- Simplifies the implementation
- Makes the code less type safe

Add "full" type definitions

- Types for all functions

Adding Types to Decorators

```
function log(
  target: any,
  context: any
) {
  const resultMethod = async function (this: any, ...args: any) {
    console.log(`@log - ...`)
    ...
  }

  return resultMethod
}
```

Decorator's Context

```
interface ClassMethodDecoratorContext<
    This = unknown,
    Value extends (this: This, ...args: any) => any = (this: This, ...args: any)
=> any,
> {
    readonly kind: "method";
    readonly name: string | symbol;
    readonly static: boolean;
    readonly private: boolean;
    readonly access: {
        has(object: This): boolean;
        get(object: This): Value;
    };

    addInitializer(initializer: (this: This) => void): void;
}
```

Adding Types to Decorators

```
function log(
  target: any,
  context: any
) {
  const resultMethod = async function (this: any, ...args: any) {
    console.log(`@log - ...`)
    ...
  }

  return resultMethod
}
```

Adding Types to Decorators

```
function log<This, Args extends any[], Return>(
  target: any,
  context: any
) {
  const resultMethod = async function (this: any, ...args: any) {
    console.log(`@log - ...`)
    ...
  }

  return resultMethod
}
```

Adding Types to Decorators

```
function log<This, Args extends any[], Return>(
  target: (this: This, ...args: Args) => Return,
  context: any
) {
  const resultMethod = async function (this: any, ...args: any) {
    console.log(`@log - ...`)
    ...
  }

  return resultMethod
}
```

Adding Types to Decorators

```
function log<This, Args extends any[], Return>(
  target: (this: This, ...args: Args) => Return,
  context: ClassMethodDecoratorContext<


  >,
) {
  const resultMethod = async function (this: any, ...args: any) {
    console.log(`@log - ...`)
    ...
  }

  return resultMethod
}
```

Adding Types to Decorators

```
function log<This, Args extends any[], Return>(
  target: (this: This, ...args: Args) => Return,
  context: ClassMethodDecoratorContext<
    This,

  >,
) {
  const resultMethod = async function (this: any, ...args: any) {
    console.log(`@log - ...`)
    ...
  }

  return resultMethod
}
```

Adding Types to Decorators

```
function log<This, Args extends any[], Return>(
  target: (this: This, ...args: Args) => Return,
  context: ClassMethodDecoratorContext<
    This,
    (this: This, ...args: Args) => Return
  >,
) {
  const resultMethod = async function (this: any, ...args: any) {
    console.log(`@log - ...`)
    ...
  }

  return resultMethod
}
```

Adding Types to Decorators

```
function log<This, Args extends any[], Return>(
  target: (this: This, ...args: Args) => Return,
  context: ClassMethodDecoratorContext<
    This,
    (this: This, ...args: Args) => Return
  >,
): (this: This, ...args: Args) => Return {
  const resultMethod = async function (this: any, ...args: any) {
    console.log(`@log - ...`)
    ...
  }

  return resultMethod
}
```

Adding Types to Decorators

```
function log<This, Args extends any[], Return>(
  target: (this: This, ...args: Args) => Return,
  context: ClassMethodDecoratorContext<
    This,
    (this: This, ...args: Args) => Return
  >,
): (this: This, ...args: Args) => Return {
  const resultMethod = async function (this: This, ...args: Args): Return {
    console.log(`@log - ...`)
    ...
  }

  return resultMethod
}
```

Restricting Types

```
function log<This, Args extends any[], Return>(
  target: (this: This, ...args: Args) => Return,
  context: ClassMethodDecoratorContext<
    This,
    (this: This, ...args: Args) => Return
  >,
): (this: This, ...args: Args) => Return {
  const resultMethod = async function (this: This, ...args: Args): Return {
    console.log(`@log - ...`)
    ...
  }

  return resultMethod
}
```

Restricting Types

```
function log<This, Args extends number[], Return>(
  target: (this: This, ...args: Args) => Return,
  context: ClassMethodDecoratorContext<
    This,
    (this: This, ...args: Args) => Return
  >,
): (this: This, ...args: Args) => Return {
  const resultMethod = async function (this: This, ...args: Args): Return {
    console.log(`@log - ...`)
    ...
  }

  return resultMethod
}
```

Demo

Add types to decorators

Start with simpler cases

Summary

What are decorators?

- What are they used for
- Types of decorators
- History of decorators

How to write method decorators

- TypeScript 5.0+

Summary - Method Decorator

```
function myDecorator(target, context) {
  return function (this, ...args) {
    return 'Hello world!'
  }
}

class GitHubClient {
  @myDecorator
  public async getRepos() {
    console.log('Getting GitHub repos...')
    ...
  }
}
```

Summary - Logging Decorator

```
// Write a log message before and after a target method execution
function log(target, context) {
  return function (this, ...args) {
    console.log(`Calling method at ${new Date().toISOString()}`)
    const result = target.apply(this, args)
    console.log(`Method finished at ${new Date().toISOString()}`)
    return result
  }
}
```

Summary - Multiple Decorators

```
class GitHubClient {

  @time('get-report-info.duration')
  @retry(1000, 3)
  getRepoInfo(repo: string) {
    …
  }
}
```

Summary - Multiple Decorators

```
class GitHubClient {

  @time('get-report-info.duration')
  @retry(1000, 3)
  getRepoInfo(repo: string) {
    …
  }
}
```

Summary - Decorator Factory

```
// Decorator factory
function log(logLevel: LogLevel) {
  // Decorator function
  return function (target: any, context: any) {
    // Decorated method
    const resultMethod = async function (this, ...args) {
      logger.log(`Running method ${context.name}`, logLevel)

      const res = target.apply(this, args)

      logger.log(`Finished method ${context.name}`, logLevel)
      return res
    }
    return resultMethod
  }
}
```

Summary - Decorator Patterns

Class method as decorator

Decorators composition

Collecting data in decorators

Up Next:

Implementing Advanced Decorators