Dev 101: TypeScript Legacy (Pre-v5.0) Decorators

ALEX MUTURI
AUG 01, 2025

1. Foundation: The Decorator Pattern

The **Decorator Pattern** is a structural design pattern that attaches new behaviors objects by placing them inside special wrapper objects, called decorators. This all you to add functionality without modifying the original class. To jump straight to code: [github repo]

```
// Classical Decorator Pattern Example
class Coffee {
  cost() { return 5; }
}
class MilkDecorator {
  constructor(private coffee: Coffee) {}
  cost() { return this.coffee.cost() + 1; }
}
const milkCoffee = new MilkDecorator(new Coffee());
console.log(milkCoffee.cost()); // 6
```

TypeScript legacy decorators are inspired by this concept: they let you **add behavi metadata** to classes, methods, and properties in a declarative, compile-time mann

Thanks for reading Unstacked Labs Newsletter! Subscribe for free to receive new posts and support my work.

Type your email... Subscribe

2. Pre-v5.0 Decorators: Basics & Parameters

For TypeScript versions lesser than v5.0, decorators are supported using an **experimental implementation** based on an early version of a JavaScript proposal f the TC39 committee — the team that evolves the JavaScript language. To enable experimental support for decorators, you must enable the `experimentalDecorators` compiler option either on the command line or in y `tsconfig.json`.

> Find Typescript 5.0+ article: [Typescript 5.0+ Decorators]()

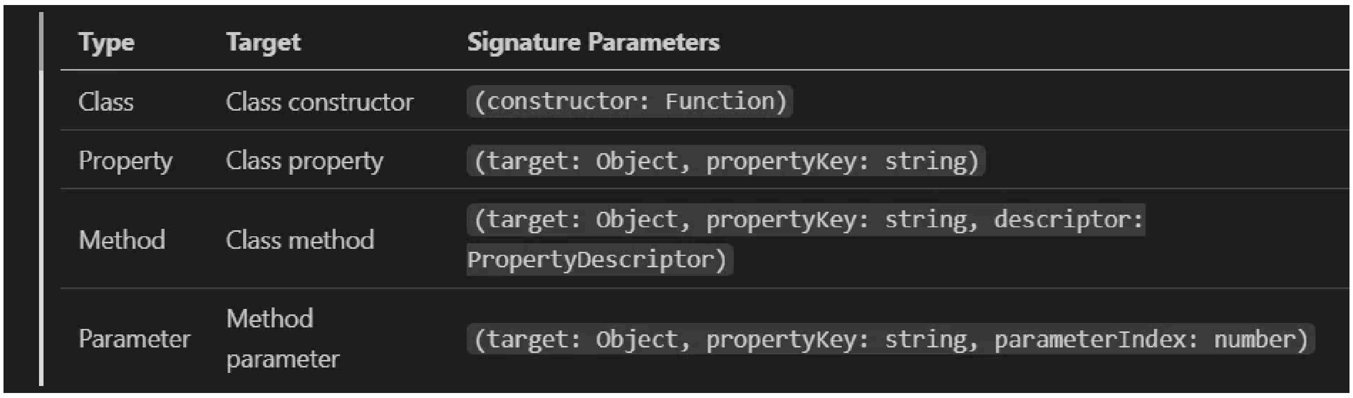
To use decorators in Pre-v5.0 TypeScript, enable them in `tsconfig.json`:

```
{
  "target": "ES2017",
  "module": "commonjs",
  "experimentalDecorators": true,
  "emitDecoratorMetadata": true
}
```

Or using the command line:

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

2.1 Types of Decorators & Their Parameters



Type	Target	Signature Parameters
Class	Class constructor	`(constructor: Function)`
Property	Class property	`(target: Object, propertyKey: string)`
Method	Class method	`(target: Object, propertyKey: string, descriptor: PropertyDescriptor)`
Parameter	Method parameter	`(target: Object, propertyKey: string, parameterIndex: number)`

Parameter Details

- **Class Decorator**: Receives the class constructor.
- **Property Decorator**: Receives the prototype and property name. Cannot modi descriptor directly.
- **Method Decorator**: Receives the prototype, method name, and descriptor (lets override/wrap the method).
- **Parameter Decorator**: Receives the prototype, method name, and parameter index.

```
// Legacy API Examples
function MyClassDecorator(constructor: Function) {
  console.log('Class created:', constructor.name);
}

function MyPropertyDecorator(target: Object, propertyKey: string) {
  console.log('Property:', propertyKey);
}

function MyMethodDecorator(target: Object, propertyKey: string,
descriptor: PropertyDescriptor) {
  const original = descriptor.value;
  descriptor.value = function (...args: any[]) {
    console.log(`Calling ${propertyKey}`, args);
    return original.apply(this, args);
  };
  return descriptor;
}

function MyParamDecorator(target: Object, propertyKey: string,
parameterIndex: number) {
  console.log(`Parameter ${parameterIndex} in ${propertyKey}`);
}
```

Example Recap:

```
class Example {
  @MyPropertyDecorator
  someProperty: string;

  @MyMethodDecorator
  someMethod(@MyParamDecorator msg: string) {}
}
```

> Decorators execute at **compile/design time**. You can modify behavior or attach annotations used by frameworks.

3. Reflection & Metadata

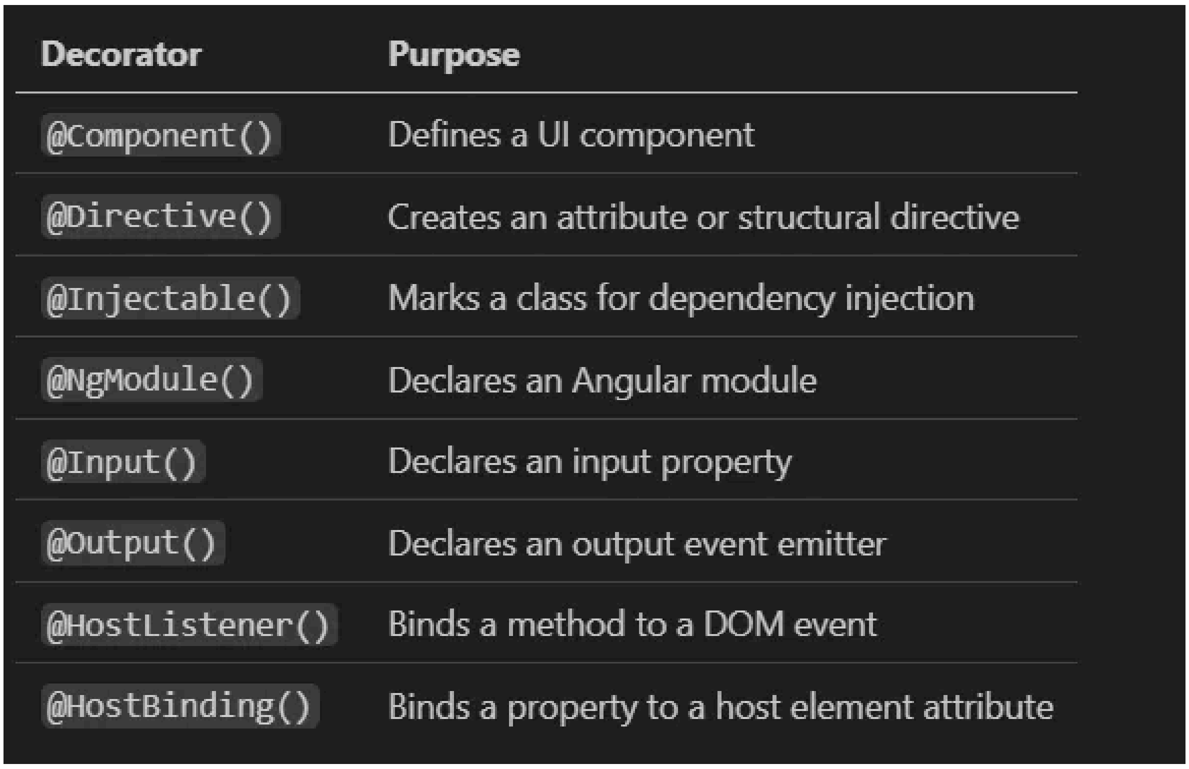
To read design-time types at runtime, use **reflect-metadata**:

1. Install: `npm install reflect-metadata`
2. Import once: `import 'reflect-metadata';`
3. Enable `emitDecoratorMetadata` in `tsconfig.json`.

```
// Retrieve the type of a property
Reflect.getMetadata('design:type', target, propertyKey);
```

Reflection lets decorators inspect types, useful for form builders, DI, or validation

4. Angular Built-in Decorators



Decorator	Purpose
`@Component()`	Defines a UI component
`@Directive()`	Creates an attribute or structural directive
`@Injectable()`	Marks a class for dependency injection
`@NgModule()`	Declares an Angular module
`@Input()`	Declares an input property
`@Output()`	Declares an output event emitter
`@HostListener()`	Binds a method to a DOM event
`@HostBinding()`	Binds a property to a host element attribute

> Angular started using decorators **long before** TC39 finalized the spec — and re
> on TypeScript's experimental decorators. Although under the hood Angular rel
> on TypeScript's standard compiler, it has its own **Angular compiler** (**ngc**) that
> processes the decorators and generates code for templates, DI, etc. See The Da
> of Decorated Classes: Inside the Angular Compiler. Hence, `ngc` can be said to l
> extension TypeScript compiler which knows how to "execute" Angular decorat
> applying their effects to the decorated classes at build time.

5. Writing Custom Decorators (Legacy API)

5.1 Class Decorator

```
function LogClass(constructor: Function) {
  console.log('Class created:', constructor.name);
}
```

```
@LogClass
class MyService {}
```

5.2 Method Decorator

```
function LogMethod(target: any, propertyKey: string, descriptor:
PropertyDescriptor): PropertyDescriptor {
  const original = descriptor.value;
  descriptor.value = function (...args: any[]) {
    console.log(`Calling ${propertyKey} with`, args);
    return original.apply(this, args);
  };
  return descriptor;
}

class Example {
  @LogMethod
  say(message: string) {
    console.log(message);
  }
}
```

5.3 Property Decorator

```
function DefaultValue(val: any) {
  return function (target: any, propertyKey: string) {
    let value = val;
    Object.defineProperty(target, propertyKey, {
      get: () => value,
      set: v => (value = v),
    });
  };
}

class Example {
  @DefaultValue("hello")
  message: string;
}
```

5.4 Parameter Decorator

```
function MyParamDecorator(target: Object, propertyKey: string,
parameterIndex: number) {
  console.log(`Parameter ${parameterIndex} in method ${propertyKey}`);
}

class Example {
  someMethod(@MyParamDecorator msg: string) {}
}
```

6. Decorator Factories & Composition

6.1 Decorator Factories

A **decorator factory** is a function that returns a decorator, allowing you to pass arguments/configuration.

```
function LogWithPrefix(prefix: string) {
  return function (target: any, propertyKey: string, descriptor:
PropertyDescriptor): PropertyDescriptor {
    const original = descriptor.value;
    descriptor.value = function (...args: any[]) {
      console.log(`${prefix}: ${propertyKey}`, args);
      return original.apply(this, args);
    };
    return descriptor;
  };
}

class Example {
  @LogWithPrefix('DEBUG')
  say(message: string) {
    console.log(message);
  }
}
```

6.2 Decorator Composition

Decorator composition means stacking multiple decorators on a single target. Decorators are applied **bottom to top** (the last decorator in code is applied first).

```
function Log(target: any, propertyKey: string, descriptor:
PropertyDescriptor): PropertyDescriptor {
  const original = descriptor.value;
  descriptor.value = function (...args: any[]) {
    console.log('Calling', propertyKey);
    return original.apply(this, args);
  };
  return descriptor;
}

function Cache(target: any, propertyKey: string, descriptor:
PropertyDescriptor): PropertyDescriptor {
  const cache = new Map<string, any>();
  const original = descriptor.value;
  descriptor.value = function (...args: any[]) {
    const cacheKey = JSON.stringify(args);
    if (cache.has(cacheKey)) {
      return cache.get(cacheKey);
    }
    const result = original.apply(this, args);
    cache.set(cacheKey, result);
    return result;
  };
  return descriptor;
}

class MathService {
  @Log
  @Cache
  expensiveOperation(x: number) {
    return x * x; // pretend this is slow
  }
}
```

7. Real-World Use Cases

7.1 Permission Check

```
function RequirePermission(permission: string) {
  return function (target: any, propertyKey: string, descriptor: PropertyDescriptor): PropertyDescriptor {
    const original = descriptor.value;
    descriptor.value = function (...args: any[]) {
      const svc = inject(PermissionService);
      if (!svc.hasPermission(permission)) {
        throw new Error(`Missing ${permission}`);
      }
      return original.apply(this, args);
    };
    return descriptor;
  };
}

class UserService {
  @RequirePermission('admin')
  deleteUser(id: string) {
    // Only admins can call this
  }
}
```

7.2 Caching

```
const cache = new Map<string, any>();
function CacheResult(target: any, propertyKey: string, descriptor: PropertyDescriptor): PropertyDescriptor {
  const original = descriptor.value;
  descriptor.value = function (...args: any[]) {
    const ckey = `${propertyKey}-${JSON.stringify(args)}`;
    if (cache.has(ckey)) return cache.get(ckey);
    const res = original.apply(this, args);
    cache.set(ckey, res);
    return res;
  };
  return descriptor;
}

class DataService {
  @CacheResult
  fetchUser(id: string) {
    // Expensive API call
  }
}
```

7.3 Analytics on Component Init

```
function WithAnalytics(eventName?: string) {
  return function (constructor: any) {
    const originalInit = constructor.prototype.ngOnInit;
    constructor.prototype.ngOnInit = function () {
      const analytics = inject(AnalyticsService);
      analytics.trackEvent('Init', { name: eventName || constructor.na
});
      originalInit?.apply(this);
    };
  };
}

@WithAnalytics('UserComponent')
class UserComponent {
  ngOnInit() {
    // Component initialization
  }
}
```

7.4 Auto-Unsubscribe

```
function AutoUnsubscribe(constructor: any) {
  const originalDestroy = constructor.prototype.ngOnDestroy;
  constructor.prototype.ngOnDestroy = function () {
    for (const k in this) {
      this[k]?.unsubscribe?.();
    }
    originalDestroy?.apply(this);
  };
}

@AutoUnsubscribe
class MyComponent {
  private subscription = new Subject();

  ngOnDestroy() {
    // Will automatically unsubscribe all subscriptions
  }
}
```

8. Bonus: Simple Signals & Observables

Bonus Simple Signal with **fromObservable** custom implementation to plug into **@AutoUnsubscribe** decorator.

8.1 SimpleSignal

```
function SimpleSignal<T>(initial: T) {
  let value = initial;
  const listeners: (() => void)[] = [];
  const signal = (() => value) as (() => T) & {
    set: (v: T) => void;
    effect: (fn: () => void) => void;
  };
  signal.set = v => { value = v; listeners.forEach(fn => fn()); };
  signal.effect = fn => { listeners.push(fn); fn(); };
  return signal;
}
```

8.2 fromObservable

```
import { Observable } from 'rxjs';
function fromObservable<T>(obs$: Observable<T>, initial: T) {
  const sig = SimpleSignal(initial);
  const sub = obs$.subscribe(val => sig.set(val));
  (sig as any).unsubscribe = () => sub.unsubscribe();
  return sig;
}
```

9. Code Lab: Debounce Decorator

Objective: Create a `@Debounce` method decorator that delays method execution.

```
function Debounce(ms: number) {
    return function (target: any, propertyKey: string, descriptor:
PropertyDescriptor): PropertyDescriptor {
        console.log("Inside the debounce decorator");
        const original = descriptor.value;
      let timeoutId: number;

      descriptor.value = function (...args: any[]) {
        clearTimeout(timeoutId);
        timeoutId = setTimeout(() => {
          original.apply(this, args);
        }, ms);
      };

      return descriptor;
    };
  }

  class SearchComponent {
    @Debounce(300)
    onSearch(query: string) {
      // This will only execute after 300ms of no calls
      console.log('I waited for 300 ms to do nothing.... arghhhh!');
    }
  }
```

Usage:

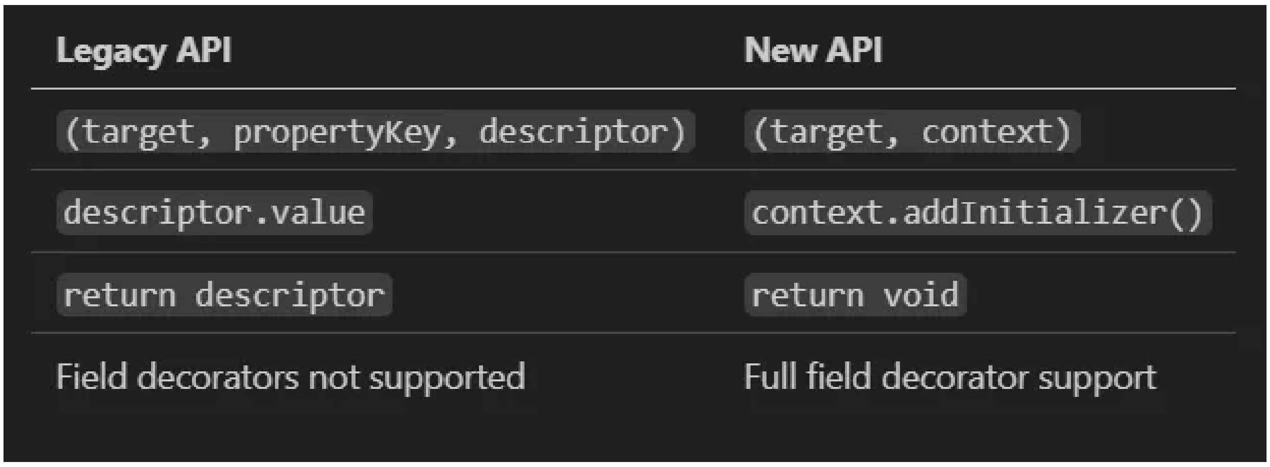
```
  // Test the debounce functionality
  const searchComponent = new SearchComponent();

  console.log('Calling onSearch multiple times quickly...');
  searchComponent.onSearch('test1');
  searchComponent.onSearch('test2');
  searchComponent.onSearch('test3');

  // The original method will only execute once after 300ms
  // because the debounce decorator delays execution and cancels
previous calls
```

10. Migration to New API

Key Differences:



Legacy API	New API
`(target, propertyKey, descriptor)`	`(target, context)`
`descriptor.value`	`context.addInitializer()`
`return descriptor`	`return void`
Field decorators not supported	Full field decorator support

Migration Example:

Legacy (Current):

```
function LogMethod(target: any, propertyKey: string, descriptor:
PropertyDescriptor) {
  const original = descriptor.value;
  descriptor.value = function (...args: any[]) {
    console.log(`Calling ${propertyKey}`, args);
    return original.apply(this, args);
  };
  return descriptor;
}
```

New API (Future):

```
function LogMethod<T extends object>(
  target: undefined,
  context: ClassMethodDecoratorContext<T, (this: T, ...args: any[]) =>
any>
): void {
  const methodName = context.name;
  context.addInitializer(function (this: T) {
    const originalMethod = this[methodName as keyof T] as (this: T,
...args: any[]) => any;
    (this as any)[methodName] = function (this: T, ...args: any[]) {
      console.log(`Calling ${String(methodName)}`, args);
      return originalMethod.call(this, ...args);
    };
  });
}
```

11. Best Practices & Summary

- **Keep decorators small and single-purpose**
- **Prefer decorator factories** for configurability
- **Document execution order** if it matters
- **Understand evaluation and application order**: decorators are applied bottom top
- **Use reflection** for advanced scenarios (type inspection, DI, validation)
- **Leverage composition** to separate concerns and reuse logic
- **Consider upgrading** to new API for TypeScript 5.0+ projects
- **Test thoroughly** when migrating between APIs

🎉 You now have a complete advanced reference for TypeScript legacy decorators, including real-world use cases, composition, and migration guidance to the new A Review remarks are always welcome.

Happy decorating.

Thanks for reading Unstacked Labs Newsletter! Subscribe for free to receive new posts and support my work.

Type your email... Subscribe

← Previous Next

Discussion about this post

Comments Restacks

Write a comment...

© 2025 Unstacked Labs · Privacy · Terms · Collection notice
Substack is the home for great culture