Topic 4: Decorators (Class, Method, Property, and Parameter Decorators with Examples)

1. Problem Statement

SkyFleet's Delivery Drones Challenge

You're the lead engineer at **SkyFleet**, managing a large fleet of autonomous delivery drones.
Business is booming, but new requirements keep coming:

- The **compliance team** wants every drone to **log its flight path** for audits.
- The **safety team** needs to **mark certain drones as "priority"** for extra security checks.
- The **analytics team** wants to **measure how long each delivery takes** to optimize routes.
- The **customer team** wants to **tag drones delivering to VIP customers** for special handling.

The problem:
How do you add these extra features to your drone classes and methods **without cluttering or rewriting the core drone code** every time a new requirement appears?

Expected outcome:

- A clean, reusable way to "attach" extra behaviors to drones, flights, or deliveries.
- Maintainable code where core drone logic stays focused and unchanged.
- The ability to add or remove features easily as business needs evolve.

2. Learning Objectives

By the end of this tutorial, you will be able to:

- Understand what decorators are and why they are useful.
- Use class, method, property, and parameter decorators in TypeScript.
- Apply decorators to add logging, tagging, timing, and marking to your drone management system.

3. Concept Introduction with Analogy

Stickers on Luggage Analogy

Imagine each drone is like a suitcase at an airport:

- You can put **stickers** on a suitcase to add information or instructions (e.g., “Fragile,” “Priority,” “Scan for Security”).
- The sticker doesn’t change what’s inside the suitcase but changes how it’s handled.

Decorators in TypeScript are like these stickers:

- You can “stick” extra instructions or behaviors onto classes, methods, or properties.
- The main drone code stays clean, but you can easily add or remove features as needed.

---

4. What Are Decorators?

A **decorator** is a special function in TypeScript that can be attached to a class, method, property, or parameter to **modify, extend, or annotate** its behavior at runtime.

Decorator Function Signatures

Class Decorator:

A class decorator is a function applied to the constructor of a class. It can observe, modify, or even replace the class definition

```
function MyClassDecorator(target: Function) { ... }
```

What can you do?

- Add metadata to the class.
- Replace or extend the class.
- Log when a class is created.

Method Decorator:
A method decorator is applied to a method in a class. It can modify the method’s behavior, add metadata, or wrap the method with extra logic.

```
function MyMethodDecorator(target: Object, propertyKey: string, descriptor: PropertyDescriptor) { ... }
```

- **target:** The class prototype for instance methods, or the constructor for static methods.
- **propertyKey:** The name of the method.
- **descriptor:** An object describing the method (can be used to replace or wrap the method).

What can you do?

- Log calls to the method.
- Change how the method works (wrap/replace it).
- Add metadata or validation.

Property Decorator:
A property decorator is applied to a class property. It’s mainly used to add metadata or modify how the property is handled

```
function MyPropertyDecorator(target: Object, propertyKey: string) { ... }
```

- **target:** The prototype of the class for instance properties, or the constructor for static properties.
- **propertyKey:** The name of the property.

What can you do?

- Add metadata to the property (e.g., mark as “priority”).
- Use with libraries/frameworks for validation or serialization.

Parameter Decorator:
A parameter decorator is applied to a parameter in a method’s signature. It’s typically used to add metadata about the parameter

```
function MyParameterDecorator(target: Object, propertyKey: string | symbol, parameterIndex: number) { ..
```

- **target:** The prototype of the class.
- **propertyKey:** The name of the method.
- **parameterIndex:** The index of the parameter in the method’s parameter list.

What can you do?

- Mark parameters for validation or special handling.
- Add metadata for frameworks or libraries.

Where Can You Use Decorators?

- **Class Decorator:** Alters or annotates the class itself.
- **Method Decorator:** Alters or wraps a method.
- **Property Decorator:** Adds metadata or changes property behavior.
- **Parameter Decorator:** Adds metadata to a method parameter.
- **Parameter and property decorators** run first, then **method/accessor decorators**, then **class decorators**.
- When stacking, decorators are applied from bottom to top (the decorator closest to the method/property runs first).

Decorator Factories

To pass arguments to a decorator, use a decorator factory-a function that returns the actual decorator.

```
function Priority(level: string) {
return function(target: any, propertyKey: string) {
  Reflect.defineMetadata("priority", level, target, propertyKey);
};
}
```

4. Step-by-Step Data Modeling

Here’s a basic drone class:

```
class Drone {
 constructor(public id: string) {}
 deliverPackage(destination: string) {
   console.log(`Drone ${this.id} delivering to ${destination}.`);
 }
}
```

5. Live Code Walkthrough

Method Decorator: Adding Logging

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

class DeliveryDrone extends Drone {
 @Log
 deliverPackage(destination: string) {
   super.deliverPackage(destination);
 }
}
```

- The `@Log` decorator adds a logging sticker to the `deliverPackage` method.

Property Decorator: Marking Priority Drones

```
function Priority(target: any, propertyKey: string) {
 Reflect.defineMetadata("priority", true, target, propertyKey);
}

class InspectionDrone extends Drone {
 @Priority
 needsExtraCheck: boolean = true;
}
```

- The `@Priority` decorator sticks a “priority” label on the `needsExtraCheck` property.

Method Decorator: Timing Deliveries

```
function Timed(target: any, propertyKey: string, descriptor: PropertyDescriptor) {
 const original = descriptor.value;
 descriptor.value = function (...args: any[]) {
   const start = Date.now();
   const result = original.apply(this, args);
   const end = Date.now();
   console.log(`[TIMER] ${propertyKey} took ${end - start}ms`);
   return result;
 };
}

class AnalyticsDrone extends Drone {
 @Timed
 deliverPackage(destination: string) {
   // Simulate delivery delay
   for (let i = 0; i < 1e7; i++) {}
   super.deliverPackage(destination);
 }
}
```

- The `@Timed` decorator adds a sticker that measures how long the delivery takes.

Parameter Decorator: Marking VIP Deliveries

```
function MarkVIP(target: any, propertyKey: string, parameterIndex: number) {
  // Could attach metadata or validation for VIP deliveries
}

class CustomerDrone extends Drone {
  deliverPackage(@MarkVIP destination: string) {
    super.deliverPackage(destination);
  }
}
```

- The `@MarkVIP` decorator tags the delivery destination parameter as VIP.

6.Challenge

Your Turn!

- Write an `@Audit` class decorator that logs when a drone instance is created.

- Apply it to a new `AuditDrone` class.

7. Quick Recap & Key Takeaways

- Decorators are like stickers that add behavior or metadata to classes, methods, properties, or parameters.
- They help keep core code clean and focused.
- You can easily add or remove features as business needs change.

8. (Optional) Programmer's Workflow Checklist

- Identify repeated or cross-cutting behaviors (logging, tagging, timing).
- Write decorators encapsulating those behaviors.
- Apply decorators to classes, methods, or properties as needed.
- Keep your core classes focused on their primary responsibilities.

9. Coming up next:

Explore **Design Patterns**!
Learn how to organize your drone fleet's code with proven blueprints like Singleton (one control tower), Factory (building new

drones), Observer (tracking deliveries), and Strategy (choosing the best delivery route).