Decorators in TypeScript

Decorators in TypeScript are special functions that can be attached to classes, methods, properties, or parameters to modify or enhance their behavior. They are a powerful feature used mainly in frameworks like **Angular**, where they play a significant role in defining metadata for components, services, and modules.

Types of Decorators

1. **Class Decorators**
2. **Method Decorators**
3. **Accessor Decorators**
4. **Property Decorators**
5. **Parameter Decorators**

How Decorators Work

Decorators are functions that are invoked with the decorated target as an argument. When you enable the `experimentalDecorators` option in `tsconfig.json`, TypeScript allows the use of decorators.

```
{
  "compilerOptions": {
    "experimentalDecorators": true
  }
}
```

1. Class Decorators

A **class decorator** is applied to the class declaration and is executed once when the class is defined.

Syntax

```
function ClassDecorator(target: Function, context: ClassDecoratorContext) {
  // Perform actions or modifications
}
```

Parameters

- `target`: The class constructor.
- `context`: Provides metadata such as `kind` and `name`.

Example

```
function LogClass(target: Function, context: ClassDecoratorContext) {
  console.log(`Class Name: ${context.name}`);
}

@LogClass
class MyClass {
  constructor() {
    console.log("Instance created");
  }
}

// Output:
// Class Name: MyClass
```

---

2. Method Decorators

A **method decorator** is applied to a method of a class. It can modify the behavior of the method.

Syntax

```
function MethodDecorator(
  target: Object,
  propertyKey: string | symbol,
  descriptor: PropertyDescriptor,
  context: MethodDecoratorContext
) {
  // Modify or enhance the method
}
```

Parameters

- `target`: The prototype of the class.
- `propertyKey`: The name of the method.
- `descriptor`: The method's property descriptor.
- `context`: Metadata about the method.

Example

```
function LogMethod(
  target: Object,
  propertyKey: string | symbol,
  descriptor: PropertyDescriptor,
  context: MethodDecoratorContext
) {
  const originalMethod = descriptor.value;

  descriptor.value = function (...args: any[]) {
    console.log(`Calling ${String(propertyKey)} with args: ${args}`);
    return originalMethod.apply(this, args);
  };
}
```

```
class MyClass {
  @LogMethod
  greet(name: string) {
    return `Hello, ${name}`;
  }
}

const obj = new MyClass();
console.log(obj.greet("John"));
// Output:
// Calling greet with args: John
// Hello, John
```

---

3. Accessor Decorators

An **accessor decorator** is applied to a getter or setter. It can modify their behavior or log metadata.

Syntax

```
function AccessorDecorator(
  target: Object,
  propertyKey: string | symbol,
  descriptor: PropertyDescriptor,
  context: AccessorDecoratorContext
) {
  // Modify or enhance the accessor
}
```

Parameters

- `target`: The prototype of the class.
- `propertyKey`: The name of the accessor.
- `descriptor`: The property descriptor of the accessor.
- `context`: Metadata about the accessor.

Example

```
function LogAccessor(
  target: Object,
  propertyKey: string | symbol,
  descriptor: PropertyDescriptor,
  context: AccessorDecoratorContext
) {
  const originalGet = descriptor.get;

  descriptor.get = function () {
    console.log(`Getting value of ${String(propertyKey)}`);
    return originalGet?.call(this);
  };
}

class MyClass {
  private _value: number = 0;

  @LogAccessor
```

```
  get value() {
    return this._value;
  }

  set value(val: number) {
    this._value = val;
  }
}

const obj = new MyClass();
obj.value = 42;
console.log(obj.value);
// Output:
// Getting value of value
// 42
```

---

4. Property Decorators

A **property decorator** is applied to a class property. It does not directly modify the value but can add metadata.

Syntax

```
function PropertyDecorator(
  target: Object,
  propertyKey: string | symbol,
  context: PropertyDecoratorContext
) {
  // Add metadata or validations
}
```

Parameters

- `target`: The prototype of the class.
- `propertyKey`: The name of the property.
- `context`: Metadata about the property.

Example

```
function LogProperty(target: Object, propertyKey: string | symbol) {
  console.log(`Property ${String(propertyKey)} has been decorated`);
}

class MyClass {
  @LogProperty
  name: string;

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

// Output:
// Property name has been decorated
```

---

5. Parameter Decorators

A **parameter decorator** is applied to a method parameter. It can capture metadata about the parameter.

Syntax

```
function ParameterDecorator(
  target: Object,
  propertyKey: string | symbol,
  parameterIndex: number,
  context: ParameterDecoratorContext
) {
  // Capture metadata about the parameter
}
```

Parameters

- `target`: The prototype of the class.
- `propertyKey`: The name of the method.
- `parameterIndex`: The index of the parameter in the method's parameter list.
- `context`: Metadata about the parameter.

Example

```
function LogParameter(
  target: Object,
  propertyKey: string | symbol,
  parameterIndex: number
) {
  console.log(
    `Parameter at index ${parameterIndex} in method ${String(
      propertyKey
    )} has been decorated`
  );
}

class MyClass {
  greet(@LogParameter name: string) {
    console.log(`Hello, ${name}`);
  }
}

const obj = new MyClass();
obj.greet("John");
// Output:
// Parameter at index 0 in method greet has been decorated
// Hello, John
```

Key Context Objects in TypeScript 5 Decorators

1. **ClassDecoratorContext**:
   - Properties: `kind` (always `"class"`), `name` (class name).
2. **MethodDecoratorContext**:

    - Properties: `kind` (always `"method"`), `name` (method name), `isStatic` (true if the method is static).
3. **AccessorDecoratorContext**:
    - Properties: `kind` (always `"accessor"`), `name` (accessor name), `isStatic`.
4. **PropertyDecoratorContext**:
    - Properties: `kind` (always `"property"`), `name` (property name).
5. **ParameterDecoratorContext**:
    - Properties: `kind` (always `"parameter"`), `name` (method name).

Use Cases of Class Decorators

1. **Add Metadata to a Class**
2. **Modify Class Behavior**
3. **Dependency Injection**
4. **Replace the Class**

Adding Metadata to a Class

Example: Storing Class Metadata

Steps

1. **Install `reflect-metadata` Package**:
    - Run the following command to add the `reflect-metadata` library:

      ```
      npm install reflect-metadata
      ```

2. **Enable Metadata Support in TypeScript**:
    - Update your `tsconfig.json` to include the following options:

      ```
      {
        "compilerOptions": {
      ```

```
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true
  }
}
```

3. **Import `reflect-metadata` in Your File**:
   - Add the following import statement at the top of the file using `Reflect`:

```
import 'reflect-metadata';
```

Example

```
import 'reflect-metadata';
function Entity(tableName: string) {
  return function (constructor: Function) {
    Reflect.defineMetadata('tableName', tableName, constructor);
    console.log(`${constructor.name} is mapped to table: ${tableName}`);
  };
}

@Entity('users')
class User {
  constructor(public id: number, public name: string) {}
}

// Output: User is mapped to table: users
```

- The `@Entity` decorator adds a table name as metadata to the `User` class.

3. Modifying Class Behavior

Example: Adding New Methods

```
function AddTimestamp<T extends { new (...args: any[]): {} }>(constructor: T)
{
  return class extends constructor {
    createdAt = new Date();
    getTimestamp() {
      return this.createdAt.toISOString();
    }
  };
}

@AddTimestamp
class Product {
  constructor(public name: string, public price: number) {}
}
```

```
// Use type assertion to access new properties
const product = new Product('Laptop', 1500) as Product & { createdAt: Date;
getTimestamp(): string };

console.log(product.createdAt); // Logs the timestamp when the product was
created
console.log(product.getTimestamp()); // Logs the timestamp in ISO format
```

- The `@AddTimestamp` decorator modifies the class by adding a new property, `createdAt` and getTimestamp().

4. Dependency Injection

Example: Injecting Services

```
function Injectable(constructor: Function) {
  console.log(`${constructor.name} is now injectable.`);
}

@Injectable
class AuthService {
  login(username: string, password: string) {
    console.log(`${username} logged in.`);
  }
}

// Output: AuthService is now injectable.
const authService = new AuthService();
authService.login('Alice', '12345'); // Alice logged in.
```

Explanation:

- The `@Injectable` decorator marks the `AuthService` class for dependency injection.

5. Replacing a Class

Example: Creating a Proxy

```
function ReplaceWithProxy<T extends { new (...args: any[]): {}
}>(constructor: T) {
  return class extends constructor {
    proxyEnabled = true;
  };
}

@ReplaceWithProxy
class Order {
  constructor(public id: number, public amount: number) {}
}
```

```
const order = new Order(1, 500);
console.log(order.proxyEnabled); // true
```

- The `@ReplaceWithProxy` decorator replaces the `Order` class with a modified version that includes a `proxyEnabled` property.

---

6. Combining Multiple Class Decorators

Decorators can be stacked, and they are executed in the order of their declaration (from bottom to top).

Example: Combining Decorators

```
function Logger(constructor: any) {
  console.log(`Logger: Class ${constructor.name} is created.`);
}

function Auditable(constructor: any) {
  console.log(`Auditable: Tracking changes for ${constructor.name}.`);
}

@Auditable
@Logger
class Invoice {
  constructor(public amount: number) {}
}

// Output:
// Logger: Class Invoice is created.
// Auditable: Tracking changes for Invoice.
```

Real-World Example: Role-Based Access

Example: Authorizing Classes

```
function Authorize(role: string) {
  return function (constructor: any) {
    console.log(`${constructor.name} can only be accessed by ${role}.`);
  };
}

@Authorize('admin')
class AdminPanel {
  constructor(public adminName: string) {}
}

// Output: AdminPanel can only be accessed by admin.
```

Key Points

1. Class decorators operate on the constructor of a class.
2. They can:
    - Add metadata.
    - Modify or extend class functionality.
    - Replace the class definition.
3. Enable powerful features like dependency injection, logging, or access control.

2. Method Decorators

Method decorators are used to add additional behavior or functionality to methods in a class. They are applied to the method of a class and can be used for logging, access control, validation, or other purposes.

How Method Decorators Work

1. **Signature of a Method Decorator**:

   ```
   function (target: Object, propertyKey: string | symbol, descriptor: PropertyDescriptor): void | PropertyDescriptor
   ```

    - `target`: The prototype of the class (or the constructor if the method is static).
    - `propertyKey`: The name of the method.
    - `descriptor`: The property descriptor of the method, which contains:
        - `value`: The actual function.
        - `writable`: Whether the method can be changed.
        - `enumerable`: Whether the method appears during enumeration of object properties.
        - `configurable`: Whether the method can be reconfigured.
2. **Modifying Behavior**: The method decorator can:
    - Modify the method by replacing `descriptor.value`.
    - Access metadata about the method.
    - Add pre/post-execution logic.

Coding Example: Logging Decorator

Objective

Log method calls with details like arguments and return values.

Code

```
// Define a method decorator
```

```
function LogMethod(target: Object, propertyKey: string | symbol,
descriptor: PropertyDescriptor) {
  const originalMethod = descriptor.value; // Save the original method

  // Modify the method
  descriptor.value = function (...args: any[]) {
    console.log(`Method ${String(propertyKey)} is called with arguments:`,
args);
    const result = originalMethod.apply(this, args); // Call the original
method
    console.log(`Method ${String(propertyKey)} returned:`, result);
    return result;
  };

  return descriptor; // Return the updated descriptor
}

// Class using the decorator
class Calculator {
  @LogMethod
  add(a: number, b: number): number {
    return a + b;
  }

  @LogMethod
  multiply(a: number, b: number): number {
    return a * b;
  }
}

// Usage
const calc = new Calculator();
calc.add(5, 10); // Logs method call and result
calc.multiply(4, 3); // Logs method call and result
```

Output

```
Method add is called with arguments: [ 5, 10 ]
Method add returned: 15
Method multiply is called with arguments: [ 4, 3 ]
Method multiply returned: 12
```

Advanced Example: Access Control

Objective

Restrict access to methods based on roles.

```
// Define a method decorator for role-based access control
function RequireRole(role: string) {
  return function (target: Object, propertyKey: string | symbol,
descriptor: PropertyDescriptor) {
    const originalMethod = descriptor.value;

    descriptor.value = function (...args: any[]) {
```

```
      const userRole = (this as any).role; // Access `role` from the
instance
      if (userRole !== role) {
        throw new Error(`Access denied: User does not have the '${role}'
role.`);
      }
      return originalMethod.apply(this, args); // Call the original method
if access is allowed
    };

    return descriptor;
  };
}

// Class using the decorator
class AdminPanel {
  role: string;

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

  @RequireRole('admin')
  deleteUser(userId: number) {
    console.log(`User with ID ${userId} deleted.`);
  }
}

// Usage
const admin = new AdminPanel('admin');
admin.deleteUser(123); // Allowed

const user = new AdminPanel('user');
try {
  user.deleteUser(456); // Throws error
} catch (error) {
  console.error(error.message);
}
```

Output

```
User with ID 123 deleted.
Access denied: User does not have the 'admin' role.
```

Assignment: Access Control System Using Decorators in TypeScript

Objective:

Design an **Access Control System** for a company where:

1. Employees have roles like `Admin`, `Manager`, and `Employee`.

2. Certain actions are restricted based on roles.
3. Decorators are used to enforce these restrictions at the method level.

Problem Statement:

You are tasked with creating a system where:

- **Roles**: `Admin`, `Manager`, `Employee`.
- Actions like `viewReports`, `editReports`, and `deleteReports` should be restricted based on roles.
- Implement a decorator to validate whether the current user has the required permissions to perform the action.
- If a user tries to access a restricted method, the system should log an error and deny access.

Instructions:

1. **Create a User Model**:
   - Define a `User` class with properties like `id`, `name`, and `role`.
2. **Define Permissions**:
   - Create a mapping of roles to their allowed actions.
3. **Create an Access Control Decorator**:
   - Implement a method decorator `@Authorize` that:
     - Checks if the user's role allows the action.
     - Logs an error if the user lacks permission.
     - Executes the method if permission is granted.
4. **Test Scenarios**:
   - Create users with different roles.
   - Attempt to perform actions with and without permission.
   - Handle unauthorized access gracefully.