Here’s how you can implement the \*\*Decorator Pattern\*\* for a `Car` interface. The \*\*Decorator Pattern\*\* allows us to dynamically add functionality to objects without modifying their existing code.

---

### \*\*Code Implementation\*\*

#### 1. \*\*Interface: Car\*\*

Defines the core `assemble()` method.

```java

package decorator;

public interface Car {

void assemble();

}

```

---

#### 2. \*\*Concrete Class: BasicCar\*\*

Implements the `Car` interface as a basic car.

```java

package decorator;

public class BasicCar implements Car {

@Override

public void assemble() {

System.out.print("Basic Car");

}

}

```

---

#### 3. \*\*Abstract Decorator: CarDecorator\*\*

Implements the `Car` interface and holds a reference to a `Car` object.

```java

package decorator;

public abstract class CarDecorator implements Car {

protected Car car;

public CarDecorator(Car car) {

this.car = car;

}

@Override

public void assemble() {

this.car.assemble();

}

}

```

---

#### 4. \*\*Concrete Decorators\*\*

Add additional functionalities to the `Car`.

\*\*a. SportsCar\*\*

```java

package decorator;

public class SportsCar extends CarDecorator {

public SportsCar(Car car) {

super(car);

}

@Override

public void assemble() {

super.assemble();

System.out.print(" Adding features of Sports Car.");

}

}

```

\*\*b. LuxuryCar\*\*

```java

package decorator;

public class LuxuryCar extends CarDecorator {

public LuxuryCar(Car car) {

super(car);

}

@Override

public void assemble() {

super.assemble();

System.out.print(" Adding features of Luxury Car.");

}

}

```

---

#### 5. \*\*Main Class\*\*

Test the Decorator Pattern implementation.

```java

package decorator;

public class DecoratorPatternDemo {

public static void main(String[] args) {

// Basic Car

Car basicCar = new BasicCar();

System.out.println("Basic Car:");

basicCar.assemble();

System.out.println("\n");

// Sports Car

Car sportsCar = new SportsCar(new BasicCar());

System.out.println("Sports Car:");

sportsCar.assemble();

System.out.println("\n");

// Luxury Car

Car luxuryCar = new LuxuryCar(new BasicCar());

System.out.println("Luxury Car:");

luxuryCar.assemble();

System.out.println("\n");

// Sports and Luxury Car combined

Car sportsLuxuryCar = new SportsCar(new LuxuryCar(new BasicCar()));

System.out.println("Sports and Luxury Car:");

sportsLuxuryCar.assemble();

System.out.println();

}

}

```

---

### \*\*Program Output\*\*

```

Basic Car:

Basic Car

Sports Car:

Basic Car Adding features of Sports Car.

Luxury Car:

Basic Car Adding features of Luxury Car.

Sports and Luxury Car:

Basic Car Adding features of Luxury Car. Adding features of Sports Car.

```

---

### \*\*Explanation\*\*

1. \*\*Core Concept\*\*:

- The `Car` interface defines the `assemble()` method.

- The `BasicCar` class provides the default implementation.

2. \*\*Decorator Class\*\*:

- `CarDecorator` implements the `Car` interface and forwards the calls to the `Car` instance it wraps.

3. \*\*Concrete Decorators\*\*:

- `SportsCar` and `LuxuryCar` extend the `CarDecorator` class to add their specific functionalities.

4. \*\*Flexibility\*\*:

- You can dynamically add or combine multiple behaviors (e.g., a car that is both a luxury car and a sports car).

This implementation showcases how the \*\*Decorator Pattern\*\* allows dynamic addition of functionalities without altering the base class or using inheritance.