Decorator Pattern

A Decorator Pattern says that just **"attach a flexible additional responsibilities to an object dynamically".**

In other words, The Decorator Pattern uses composition instead of inheritance to extend the functionality of an object at runtime.

The Decorator Pattern is also known as **Wrapper.**

Decorator Pattern

A Decorator Pattern says that just **"attach a flexible additional responsibilities to an object dynamically".**

In other words, The Decorator Pattern uses composition instead of inheritance to extend the functionality of an object at runtime.

The Decorator Pattern is also known as **Wrapper.**

Advantage of Decorator Pattern

- It provides greater flexibility than static inheritance.
- It enhances the extensibility of the object, because changes are made by coding new classes.
- It simplifies the coding by allowing you to develop a series of functionality from targeted classes instead of coding all of the behavior into the object.

Usage of Decorator Pattern

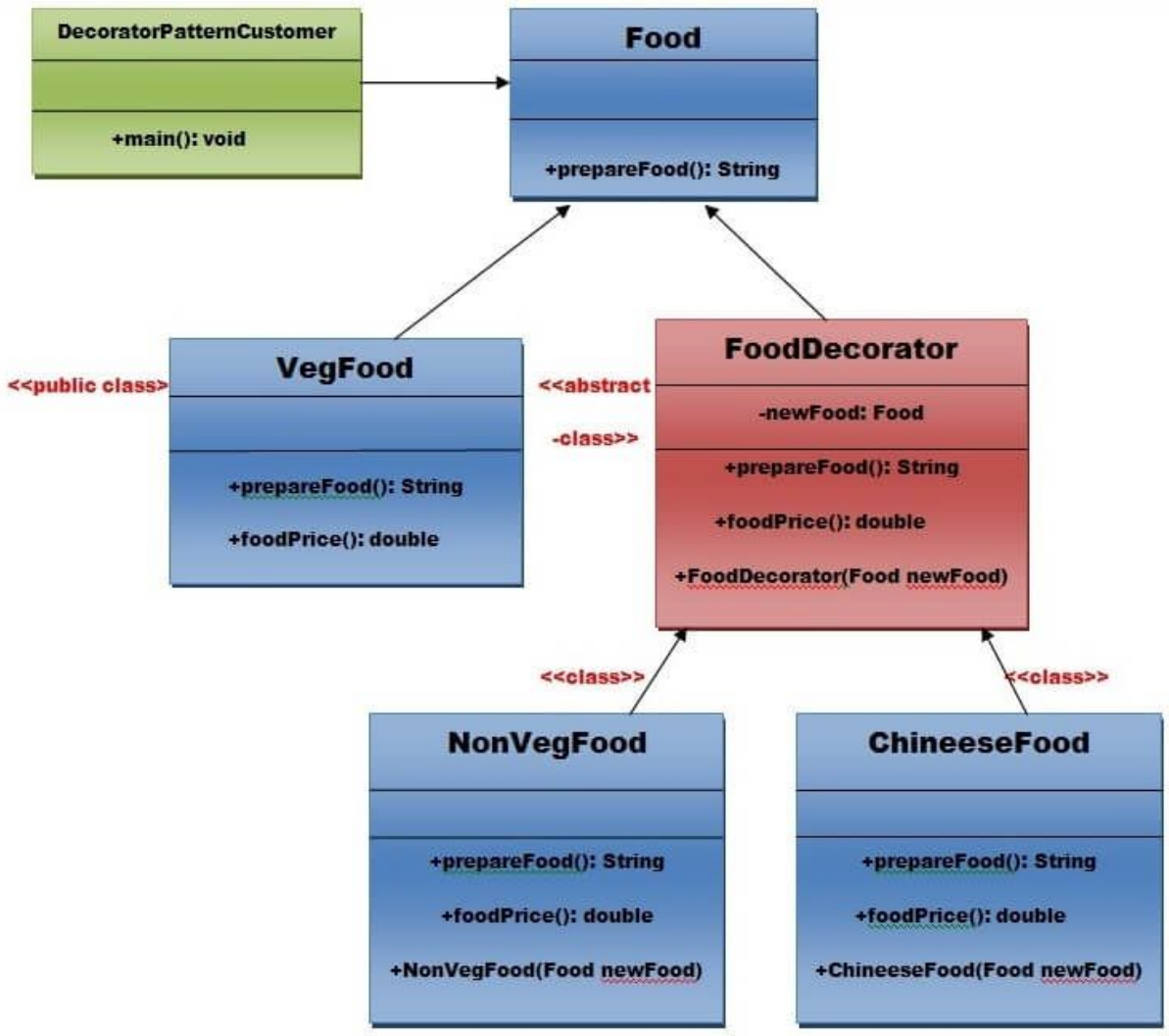
It is used:

- When you want to transparently and dynamically add responsibilities to objects without affecting other objects.
- When you want to add responsibilities to an object that you may want to change in future.
- Extending functionality by sub-classing is no longer practical

Usage of Decorator Pattern

It is used:

- When you want to transparently and dynamically add responsibilities to objects without affecting other objects.
- When you want to add responsibilities to an object that you may want to change in future.
- Extending functionality by sub-classing is no longer practical



Step 1:**Create a Food interface.**

```
1. public interface Food {
2.     public String prepareFood();
3.     public double foodPrice();
```

```
}// End of the Food interface.
public interface Food {
    public String prepareFood();
    public double foodPrice();
}// End of the Food interface.
```

Step 2: Create a **VegFood** class that will implements the **Food** interface and override its all methods.

```
public class VegFood implements Food {
    public String prepareFood(){
        return "Veg Food";
    }

    public double foodPrice(){
    return 50.0;
    }
}
```

Step 3:Create a FoodDecorator abstract class that will implements the Food interface and override it's all methods and it has the ability to decorate some more foods.

```
public abstract class FoodDecorator implements Food{
    private Food newFood;
    public FoodDecorator(Food newFood)  {
        this.newFood=newFood;
    }
    @Override
    public String prepareFood(){
        return newFood.prepareFood();
    }
    public double foodPrice(){
        return newFood.foodPrice();
    }
}

```

Step 4:Create a **NonVegFood concrete** class that will extend the **FoodDecorator** class and override it's all methods.

```
public class NonVegFood extends FoodDecorator{
```

```
2.      public NonVegFood(Food newFood) {
3.         super(newFood);
4.      }
5.      public String prepareFood(){
6.         return super.prepareFood() +" With Roasted Chiken and Chiken Curry  ";
7.      }
8.      public double foodPrice()   {
9.         return super.foodPrice()+150.0;
10.     }
11. }
```

Step 5:Create a **ChineeseFood** concrete class that will extend the **FoodDecorator** class and override it's all methods.

```
1.  public class ChineeseFood extends FoodDecorator{
2.    public ChineeseFood(Food newFood)    {
3.         super(newFood);
4.    }
5.      public String prepareFood(){
6.         return super.prepareFood() +" With Fried Rice and Manchurian  ";
7.      }
8.      public double foodPrice()   {
9.         return super.foodPrice()+65.0;
10.        }
11. }
```

Step 6:Create a **DecoratorPatternCustomer** class that will use Food interface to use which type of food customer wants means (Decorates).

File: DecoratorPatternCustomer.java

```
1.  import java.io.BufferedReader;
2.  import java.io.IOException;
3.  import java.io.InputStreamReader;
4.  public class DecoratorPatternCustomer {
5.      private static int  choice;
6.      public static void main(String args[]) throws NumberFormatException,
    IOException    {
7.         do{
```

```
8.          System.out.print("========== Food Menu ============ \n");
9.          System.out.print("          1. Vegetarian Food.   \n");
10.         System.out.print("          2. Non-Vegetarian Food.\n");
11.         System.out.print("          3. Chineese Food.       \n");
12.         System.out.print("          4. Exit                 \n");
13.         System.out.print("Enter your choice: ");
14.         BufferedReader br=new BufferedReader(new
    InputStreamReader(System.in));
15.         choice=Integer.parseInt(br.readLine());
16.         switch (choice) {
17.         case 1:{
18.                 VegFood vf=new VegFood();
19.              System.out.println(vf.prepareFood());
20.              System.out.println( vf.foodPrice());
21.            }
22.            break;
23.
24.              case 2:{
25.              Food f1=new NonVegFood((Food) new VegFood());
26.                 System.out.println(f1.prepareFood());
27.              System.out.println( f1.foodPrice());
28.         }
29.            break;
30.      case 3:{
31.             Food f2=new ChineeseFood((Food) new VegFood());
32.                   System.out.println(f2.prepareFood());
33.                  System.out.println( f2.foodPrice());
34.             }
35.            break;
36.
37.          default:{
38.            System.out.println("Other than these no food available");
39.         }
40.     return;
41.      }//end of switch
42.
43. }while(choice!=4);
```

```
44.     }
45. }
```