# Decorator pattern

**Decorator design pattern** is used to modify the functionality of an object at runtime. At the same time other instances of the same class will not be affected by this, so individual object gets the modified behavior. Decorator design pattern is one of the structural design pattern (such as Adapter Pattern, Bridge Pattern, Composite Pattern) and uses abstract classes or interface with composition to implement.

# Decorator Design Pattern

We use inheritance or composition to extend the behavior of an object but this is done at compile time and its applicable to all the instances of the class. We can't add any new functionality of remove any existing behavior at runtime – this is when Decorator pattern comes into picture.

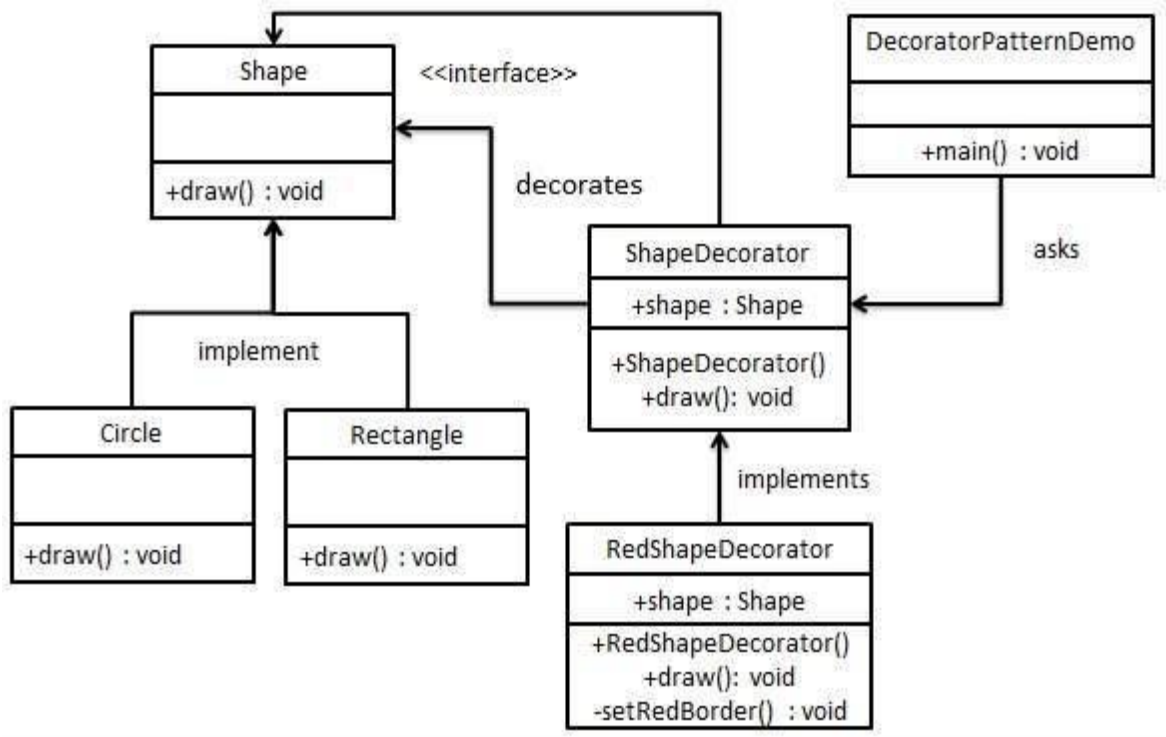
Decorator pattern attaches additional responsibilities to an object. Decorators provide a flexible alternative to subclassing for extended functionalities.

## Example 1

We're going to create a Shape interface and concrete classes implementing the Shape interface. We will then create an abstract decorator class ShapeDecorator implementing the Shape interface and having Shape object as its instance variable.

RedShapeDecorator is concrete class implementing ShapeDecorator.

DecoratorPatternDemo, our demo class will use RedShapeDecorator to decorate Shape objects.




**Step 1**

Create an interface.

*Shape.java*

```
public interface Shape {
   void draw();
}
```

**Step 2**

Create concrete classes implementing the same interface.

*Rectangle.java*

```
public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Shape: Rectangle");
   }
}
```

*Circle.java*

```
public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Shape: Circle");
   }
}
```

**Step 3**

Create abstract decorator class implementing the *Shape* interface.

*ShapeDecorator.java*

```
public abstract class ShapeDecorator implements Shape {
   protected Shape decoratedShape;

   public ShapeDecorator(Shape decoratedShape){
      this.decoratedShape = decoratedShape;
   }

   public void draw(){
      decoratedShape.draw();
   }
}
```

**Step 4**

Create concrete decorator class extending the *ShapeDecorator* class.

*RedShapeDecorator.java*

```
public class RedShapeDecorator extends ShapeDecorator {

   public RedShapeDecorator(Shape decoratedShape) {
      super(decoratedShape);
   }

   @Override
   public void draw() {
      decoratedShape.draw();
      setRedBorder(decoratedShape);
   }

   private void setRedBorder(Shape decoratedShape){
      System.out.println("Border Color: Red");
   }
}
```

**Step 5**

Use the *RedShapeDecorator* to decorate *Shape* objects.

*DecoratorPatternDemo.java*

```
public class DecoratorPatternDemo {
   public static void main(String[] args) {

      Shape circle = new Circle();

      Shape redCircle = new RedShapeDecorator(new Circle());

      Shape redRectangle = new RedShapeDecorator(new Rectangle());
      System.out.println("Circle with normal border");
      circle.draw();

      System.out.println("\nCircle of red border");
      redCircle.draw();

      System.out.println("\nRectangle of red border");
      redRectangle.draw();
   }
}
```

## Step 6

Verify the output.

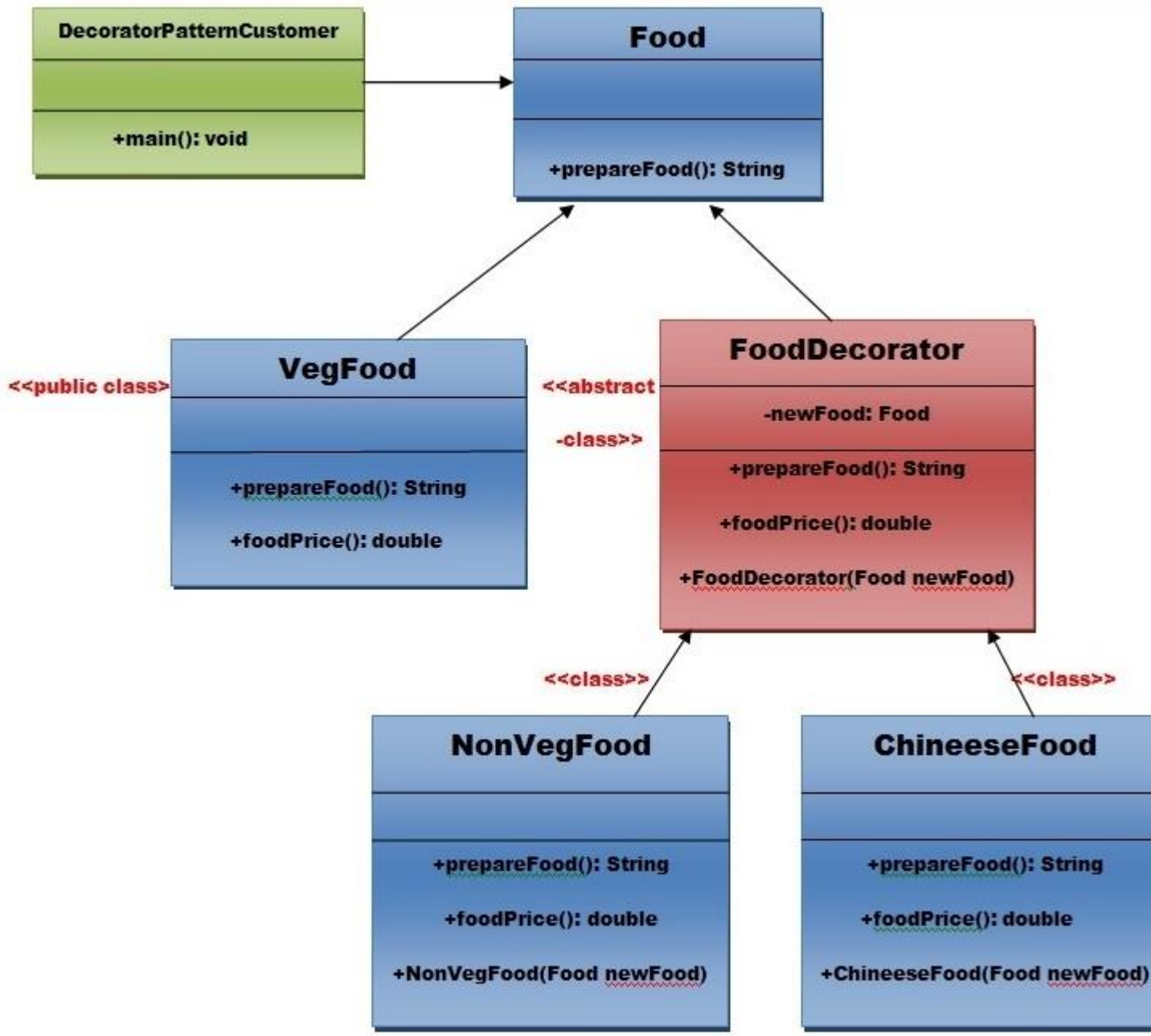
```
Circle with normal border
Shape: Circle

Circle of red border
Shape: Circle
Border Color: Red

Rectangle of red border
Shape: Rectangle
Border Color: Red
```

## Example 2

### *UML for Decorator Pattern:*



Step 1:**Create a 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{
   public NonVegFood(Food newFood) {
      super(newFood);
   }
   public String prepareFood(){
      return super.prepareFood() +" With Roasted Chiken and Chiken Curry  ";
   }
   public double foodPrice()   {
      return super.foodPrice()+150.0;
   }
}
```

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

```
public class ChineeseFood extends FoodDecorator{
  public ChineeseFood(Food newFood)    {
      super(newFood);
  }
   public String prepareFood(){
      return super.prepareFood() +" With Fried Rice and Manchurian  ";
   }
   public double foodPrice()   {
      return super.foodPrice()+65.0;
      }
}
```

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

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class DecoratorPatternCustomer {
   private static int  choice;
   public static void main(String args[]) throws NumberFormatException, IOException    {
     do{
      System.out.print("========== Food Menu ============ \n");
      System.out.print("          1. Vegetarian Food.   \n");
      System.out.print("          2. Non-Vegetarian Food.\n");
      System.out.print("          3. Chineese Food.        \n");
      System.out.print("          4. Exit                  \n");
      System.out.print("Enter your choice: ");
      BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
      choice=Integer.parseInt(br.readLine());
      switch (choice) {
      case 1:{
              VegFood vf=new VegFood();
          System.out.println(vf.prepareFood());
          System.out.println( vf.foodPrice());
         }
         break;

            case 2:{
            Food f1=new NonVegFood((Food) new VegFood());
               System.out.println(f1.prepareFood());
            System.out.println( f1.foodPrice());
     }
         break;
    case 3:{
          Food f2=new ChineeseFood((Food) new VegFood());
                System.out.println(f2.prepareFood());
               System.out.println( f2.foodPrice());
          }
```

```
            break;

          default:{
            System.out.println("Other than these no food available");
          }
       return;
        }//end of switch

}while(choice!=4);
   }
}
```

Run time output

```
========== Food Menu ============
            1. Vegetarian Food.
            2. Non-Vegetarian Food.
            3. Chineese Food.
            4. Exit
Enter your choice: 1
Veg Food
50.0
========== Food Menu ============
            1. Vegetarian Food.
            2. Non-Vegetarian Food.
            3. Chineese Food.
            4. Exit
Enter your choice: 2
Veg Food With Roasted Chiken and Chiken Curry
200.0
========== Food Menu ============
            1. Vegetarian Food.
            2. Non-Vegetarian Food.
            3. Chineese Food.
            4. Exit
Enter your choice: 3
Veg Food With Fried Rice and Manchurian
```

115.0

========== Food Menu ============

1. Vegetarian Food.
2. Non-Vegetarian Food.
3. Chineese Food.
4. Exit

Enter your choice: 4

Other than these no food available