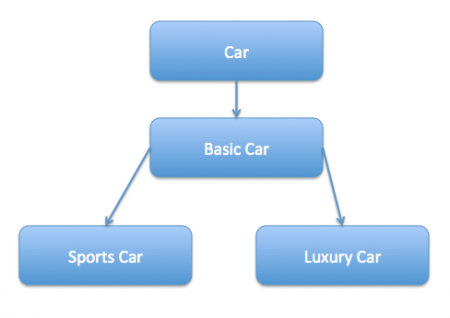
**Decorator**[**design pattern**](https://www.journaldev.com/1827/java-design-patterns-example-tutorial) is used to modify the functionality of an object at runtime. At the same time other instances of the same class will not be affected by this, so individual object gets the modified behavior. Decorator design pattern is one of the structural design pattern (such as [Adapter Pattern](https://www.journaldev.com/1487/adapter-design-pattern-java), [Bridge Pattern](https://www.journaldev.com/1491/bridge-design-pattern-java), [Composite Pattern](https://www.journaldev.com/1535/composite-design-pattern-in-java)) and uses abstract classes or interface with [composition](https://www.journaldev.com/1325/composition-in-java-example) to implement.

**Decorator Design Pattern**

We use [inheritance](https://www.journaldev.com/644/inheritance-java-example) 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.

Suppose we want to implement different kinds of cars – we can create interface Car to define the assemble method and then we can have a Basic car, further more we can extend it to Sports car and Luxury Car. The implementation hierarchy will look like below image.

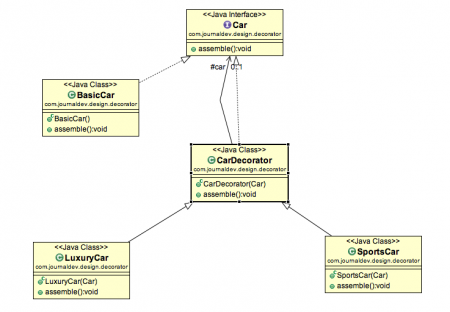
[](https://cdn.journaldev.com/wp-content/uploads/2013/07/inheritance-hierarchy.png)

But if we want to get a car at runtime that has both the features of sports car and luxury car, then the implementation gets complex and if further more we want to specify which features should be added first, it gets even more complex. Now imagine if we have ten different kind of cars, the implementation logic using inheritance and composition will be impossible to manage. To solve this kind of programming situation, we apply decorator pattern in java.

We need to have following types to implement decorator design pattern.

1. **Component Interface** – The interface or [**abstract class**](https://www.journaldev.com/1582/abstract-class-in-java) defining the methods that will be implemented. In our case Car will be the component interface.
2. package com.journaldev.design.decorator;
3. public interface Car {
4. public void assemble();
5. }
6. **Component Implementation** – The basic implementation of the component interface. We can have BasicCar class as our component implementation.
7. package com.journaldev.design.decorator;
8. public class BasicCar implements Car {
9. @Override
10. public void assemble() {
11. System.out.print("Basic Car.");
12. }
13. }
14. **Decorator** – Decorator class implements the component interface and it has a HAS-A relationship with the component interface. The component variable should be accessible to the child decorator classes, so we will make this variable protected.
15. package com.journaldev.design.decorator;
16. public class CarDecorator implements Car {
17. protected Car car;
19. public CarDecorator(Car c){
20. this.car=c;
21. }
23. @Override
24. public void assemble() {
25. this.car.assemble();
26. }
27. }
28. **Concrete Decorators** – Extending the base decorator functionality and modifying the component behavior accordingly. We can have concrete decorator classes as LuxuryCar and SportsCar.
29. package com.journaldev.design.decorator;
30. public class SportsCar extends CarDecorator {
31. public SportsCar(Car c) {
32. super(c);
33. }
34. @Override
35. public void assemble(){
36. super.assemble();
37. System.out.print(" Adding features of Sports Car.");
38. }
39. }
40. package com.journaldev.design.decorator;
41. public class LuxuryCar extends CarDecorator {
42. public LuxuryCar(Car c) {
43. super(c);
44. }
46. @Override
47. public void assemble(){
48. super.assemble();
49. System.out.print(" Adding features of Luxury Car.");
50. }
51. }

**Decorator Design Pattern – Class Diagram**

[](https://cdn.journaldev.com/wp-content/uploads/2013/07/decorator-pattern.png)

**Decorator Design Pattern Test Program**

package com.journaldev.design.test;

import com.journaldev.design.decorator.BasicCar;

import com.journaldev.design.decorator.Car;

import com.journaldev.design.decorator.LuxuryCar;

import com.journaldev.design.decorator.SportsCar;

public class DecoratorPatternTest {

public static void main(String[] args) {

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

sportsCar.assemble();

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

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

sportsLuxuryCar.assemble();

}

}

Notice that client program can create different kinds of Object at runtime and they can specify the order of execution too.

Output of above test program is:

Basic Car. Adding features of Sports Car.

\*\*\*\*\*

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

**Decorator Design Pattern – Important Points**

* Decorator design pattern is helpful in providing runtime modification abilities and hence more flexible. Its easy to maintain and extend when the number of choices are more.
* The disadvantage of decorator design pattern is that it uses a lot of similar kind of objects (decorators).
* Decorator pattern is used a lot in [Java IO](https://www.journaldev.com/942/java-io-tutorial) classes, such as [FileReader, BufferedReader](https://www.journaldev.com/867/java-read-text-file) etc.

Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class.

This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

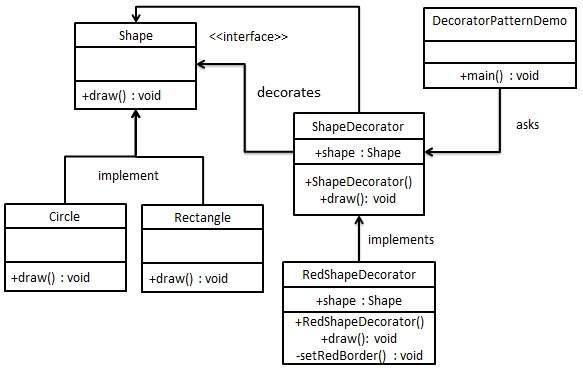
We are demonstrating the use of decorator pattern via following example in which we will decorate a shape with some color without alter shape class.

## Implementation

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

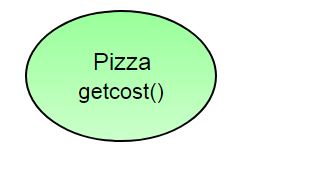
Border Color: Red

# **The Decorator Pattern | Set 2 (Introduction and Design)**

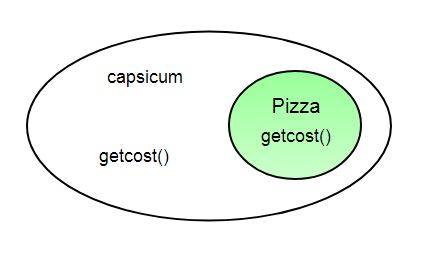
As we saw our [previous designs](https://www.geeksforgeeks.org/decorator-pattern/) using inheritance didn’t work out that well. In this article, decorator pattern is discussed for the design problem in previous set.

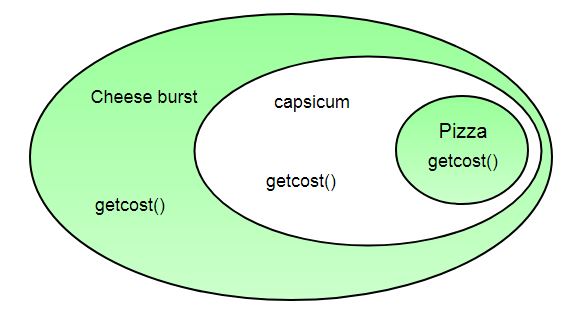
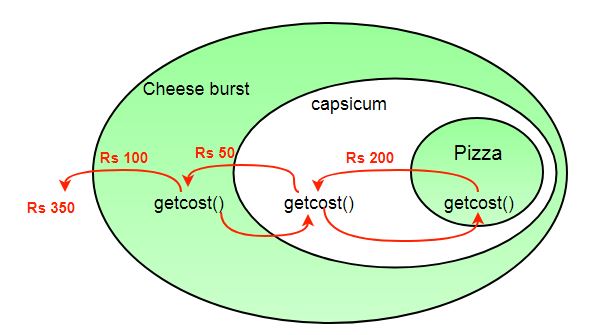
So what we do now is take a pizza and “decorate” it with toppings at runtime:

1. Take a pizza object.



1. “Decorate” it with a Capsicum object.



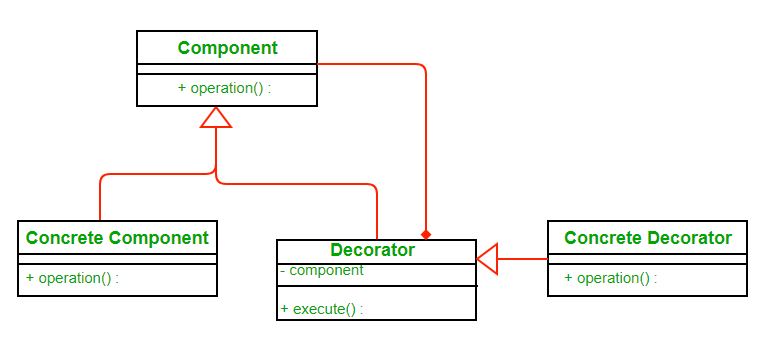
1. “Decorate” it with a CheeseBurst object.
2. Call getCost() and use delegation instead of inheritance to calculate the toppings cost.  
   

What we get in the end is a pizza with cheeseburst and capsicum toppings. Visualize the “decorator” objects  like wrappers. Here are some of the properties of decorators:

* Decorators have the same super type as the object they decorate.
* You can use multiple decorators to wrap an object.
* Since decorators have same type as object, we can pass around decorated object instead of original.
* We can decorate objects at runtime.

**Definition:**

The decorator pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

**Class Diagram:**Image src: [Wikipedia](https://upload.wikimedia.org/wikipedia/commons/thumb/e/e9/Decorator_UML_class_diagram.svg/600px-Decorator_UML_class_diagram.svg.png)

* Each component can be used on its own or may be wrapped by a decorator.
* Each decorator has an instance variable that holds the reference to component it decorates(HAS-A relationship).
* The ConcreteComponent is the object we are going to dynamically decorate.

**Advantages:**

* The decorator pattern can be used to make it possible to extend (decorate) the functionality of a certain object at runtime**.**
* The decorator pattern is an alternative to subclassing. Subclassing adds behavior at compile time, and the change affects all instances of the original class; decorating can provide new behavior at runtime for individual objects.
* Decorator offers a pay-as-you-go approach to adding responsibilities. Instead of trying to support all foreseeable features in a complex, customizable class, you can define a simple class and add functionality incrementally with Decorator objects.

**Disadvantages:**

* Decorators can complicate the process of instantiating the component because you not only have to instantiate the component, but wrap it in a number of decorators.
* It can be complicated to have decorators keep track of other decorators, because to look back into multiple layers of the decorator chain starts to push the decorator pattern beyond its true intent.