**Decorator Design Pattern**

* **Ex Sandwich prices post toppings, Pizza order Topping**

**Decorator Design Pattern in Java - Real life Example**

Decorator Pattern is one of the famous Gang of Four (GOF) structural design pattern, which provides an alternative way  of extending an object's functionality. It's different than the traditional way of adding new functionality into an object using Inheritance, since it's based on Composition and provides additional functionality at the run time, as opposite to Inheritance, which adds new functionality at compile time. Decorator design pattern is introduced by famous [Gang of Four design pattern book](http://www.amazon.com/dp/0201633612/?tag=javamysqlanta-20), almost 2 decades ago. It's a time-tested way of adding new functionalities into an object. In this Java design pattern tutorial, we will learn Decorator design pattern by using it in a Java example. This is the best way of learning design pattern, followed you try it yourself to apply in similar scenarios. Decorator pattern is one of the popular design patterns along with Factory method pattern and [Singleton Pattern](http://java67.blogspot.sg/2012/08/what-is-singleton-pattern-in-java.html), and you can see it's usage even in JDK itself. Couple of classes from java.io package e.g. BufferedInputStream, LineNumberInputStream are good example of Decorator design pattern in Java.

**Decorator design pattern in Java**

In order to show you, how to implement Decorator pattern, let me first explain requirements. Do we need to create software for calculating the price for a Sandwich, yummy... no? Since the customer can customize sandwich by asking extra cheese or extra fillings, you also need to include the cost of those items in the final price of Sandwich. Since this customization can vary a lot among different customers and offer from a shop, creating classes for different types of Sandwich with different fillings or extras e.g. BrownBreadSandWithCheese or WhiteBreaSandwitchWithCheeseAndTomato will just clutter the code with lots of endless small classes.

Now this problem looks a natural fit for applying Decorator pattern because we have a base object Sandwich, which can be decorated with extra cheese and fillings. By using [Decorator pattern](http://javarevisited.blogspot.com/2011/11/decorator-design-pattern-java-example.html), you can extend the functionality of Sandwich at runtime, based upon customer's request, which is impossible with Inheritance until you have a specific class for every possible customer request.

This is one of the reasons Why Composition is preferred over Inheritance in Object oriented design and particularly in Java. Now, let's see our class structure, We have an abstract class Sandwich, with abstract method price() and a concrete implementation class WhiteBreadSandwich, which cost $3.0.

Now, in order to provide extra cheese, which obviously incur extra cost, we are going to use *Decorator design pattern*. We have a Decorator abstract class, which will act as a base for Decorators called SandwichDecorator, and a concrete implementation of this as CheeseDecorator.

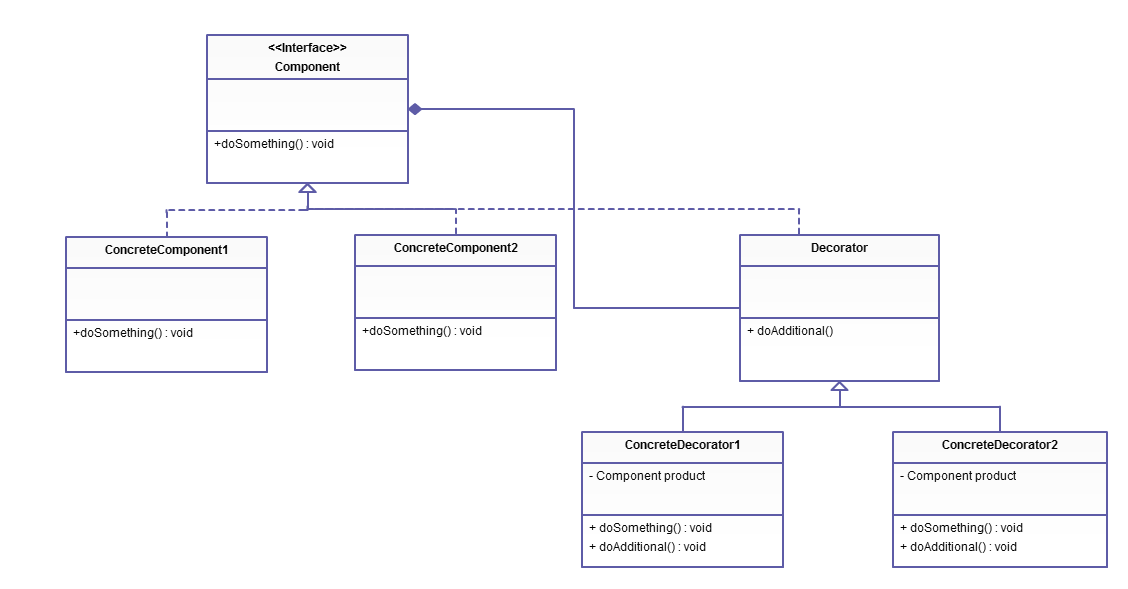
SandwichDecorator extends Sandwich, to be of the same type as original object, which is getting decorated. This is a critical requirement of Decorator pattern so that a decorated object can stand in place of original object i.e. it can be passed when a method expects original object.

Decorator adds functionality before or after delegating the task to the original object, which means in this example price of a WhilteBreadSandwich with Cheese will be calculated by the first calculating price of WhiteBreadSandwich and then the price of Cheese. Finally, I have a class called SandwichMaker, which will make a delicious sandwich for you :) I mean it will test the whole program and demonstrate how Decorator pattern adds new functionality on the fly at runtime.

I have also used BigDecimal to represent money instead of double primitive to follow best practices suggested in [Effective Java](http://www.amazon.com/dp/0321356683/?tag=javamysqlanta-20). If you don't know why read this Java mistake about using double to represent price

**UML diagram of Decorator Pattern**

Here is the UML class diagram of decorator design pattern. You can see that we have a Component interface, which is used to create ConcreteComponents. In our real world example, Sandwich is component interface and WhiteBreadSandwich is a concrete component. Now if we want to add an additional feature to Component interface, we create another interface called Decorator, which inherit Component interface, this is very important because it allows you to pass an instance of Decorator to any method which accepts an instance of Component interface. Now you can create an implementation of a decorator, which has both functionalities provided by the original interface and new feature provided by the decorator. This way you can add new features in existing class hierarchy without modifying tried and tested code. This is another reason programmers says the [composition is better than Inheritance](http://javarevisited.blogspot.com/2013/06/why-favor-composition-over-inheritance-java-oops-design.html).

[](http://3.bp.blogspot.com/-NAix3R1LC30/VI0hdX__GnI/AAAAAAAACMY/pXmco-E8jcE/s1600/Decorator%2BDesign%2BPattern%2Bin%2BJava.PNG)

**Sample Code of Decorator Design Pattern in Java**

Here is complete Java program to demonstrate how you can implement decorator pattern in Java. You can use this sample code to add more features and create new classes. If you are using Eclipse IDE, just create a Java project, select that project in the package explorer and copy the code there, it will automatically create right packages and Java classes.

**Sandwich.java**

**import** **java.math.BigDecimal**;

/\*\*

\* Base class for all types of Sandwich, cost method is abstract because

\* different sandwiches has different price.

\*

\* @author Javin Paul

\*/

**public** **abstract** **class** **Sandwich** {

**protected** String description = "Sandwich";

**public** String **getDescription**(){

**return** description;

}

**public** **abstract** BigDecimal **price**();

}

**WhiteBreadSandWich.java**

**import** **java.math.BigDecimal**;

/\*\*

\* A Concrete implementation of abstract Sandwich class, which represent a WhiteBread

\* Sandwich, whose price is 3.0$.

\*

\* @author Javin Paul

\*/

**public** **class** **WhiteBreadSandWich** **extends** Sandwich {

**public** **WhiteBreadSandWich**(String desc){

description = desc;

}

**@Override**

**public** BigDecimal **price**() {

**return** **new** **BigDecimal**("3.0");

}

}

**SandWichDecorator.java**

/\*\*

\* Base class for Decorators, this class inherit from Sandwich, so that

\* it can be of same type, which is required to pass decorators where

\* original object is expected. Later, this class will also come handy

\* to provide common functionalities to Decorators.

\*

\* @author

\*/

**public** **abstract** **class** **SandWichDecorator** **extends** Sandwich {

**@Override**

**public** **abstract** BigDecimal **price**();

}

**CheeseDecorator.java**

**import** **java.math.BigDecimal**;

/\*\*

\* A Decorator class, which adds cheese (new functionality) into Sandwich object.

\* This Decorator class modifies price() and getDescritption() method to implement

\* new behaviour.

\*

\* @author

\*/

**public** **class** **CheeseDecorator** **extends** SandWichDecorator{

Sandwich currentSandwich;

**public** **CheeseDecorator**(Sandwich sw){

currentSandwich = sw;

}

**@Override**

**public** String **getDescription**(){

**return** currentSandwich.getDescription() + ", Cheese";

}

**@Override**

**public** BigDecimal **price**() {

**return** currentSandwich.price().add(**new** BigDecimal("0.50"));

}

}

**SandwichMaker.java**

/\*\*

\* Test class to demonstrate How Decorator Pattern in Java work together. This class

\* first creates a Sandwich and decorates it with extra cheese. This is nice example

\* of how to provide new functionalities to an object at runtime using Decorator Pattern.

\*

\* @author Javain Paul

\*/

**public** **class** **SandwichMaker** {

**public** **static** **void** **main**(String args[]){

Sandwich mySandwich = **new** WhiteBreadSandWich("White bread Sandwich");

System.out.printf("Price of %s is $%.2f %n", mySandwich.getDescription(),

mySandwich.price());

//adding extra cheese using Decorator Pattter

mySandwich = **new** CheeseDecorator(mySandwich);

System.out.printf("Price of %s is $%.2f %n", mySandwich.getDescription(),

mySandwich.price());

}

}

**Output:**

Price of White bread Sandwich is $3.**00**

Price of White bread Sandwich, Cheese is $3.**50**

.

**Key things about Decorator Design Pattern**

Now, we have seen an example of decorator pattern in Java, we can quickly summarize few important things which are worth remembering while implementing or applying decorator pattern or even to answers design pattern questions like *When to use Decorator design pattern in Java*.

1) The decorator must be of the same type of object, which they are decorating. This can be achieved either by implementing the interface of the object or by extending an abstract class of the original class.

2) Decorator is based on Composition, which means it needs an original object to decorate it. This is achieved by creating a constructor on decorator class which accepts a base type of original object. e.g. in this example constructor of CheeseDecorator accepts Sandwich object. Decorator pattern is also a good example of [Open Closed design principle](http://javarevisited.blogspot.com/2011/11/great-example-of-open-closed-design.html), which is one of the key principles from Uncle Bob's SOLID design principles.

3) Decorator class adds new functionality before or after delegating the task to the original object. In this example, the price of Decorator i.e. cheese is included after calculating the price of White Bread Sandwich.

4) Remember, Decorator design pattern only affects objects at runtime, it doesn't affect the class. You should use DECORATOR PATTERN when your intent is to add new functionality at runtime (i.e. a customer order, where you only know about order details, one it placed).

5) There is *one disadvantage of Decorator pattern* as well, it adds lots of small classes in the code base, remember the overwhelming number of classes in java.io package. Though, once you know that which classes are main classes, and which are decorators, you tend to get a better understanding of overall structure. UML diagrams certain helps in this case.

That's all on **Decorator design pattern in Java and Object oriented design**. I must say, this is one of the must know design patterns for a senior Java developers, it's general purpose and has lot's of use cases as well.

Read more: <http://www.java67.com/2013/07/decorator-design-pattern-in-java-real-life-example-tutorial.html#ixzz5Oi12QYLQ>