Decorator Design Pattern

The decorator Pattern attaches additional responsibilities to an object dynamically.   
Decorator pattern provides a flexible alternative to sub classing for extended functionality

In simpler way we can explain the Decorator pattern in the following way.

You have an Obj, and a method Speak() is sending some message to the “obj”. The “Obj” is expected to return the same message as output. Now if you want change the thing that returns or changing the behaviour of the “obj” at run time, you could use decorator pattern. And this can be done at run time not at compile time. Instead of Sending messages directly to the object, from the method Speak(), we are decorating it with a wrapper. Also you can wrapper again the wrapped object.

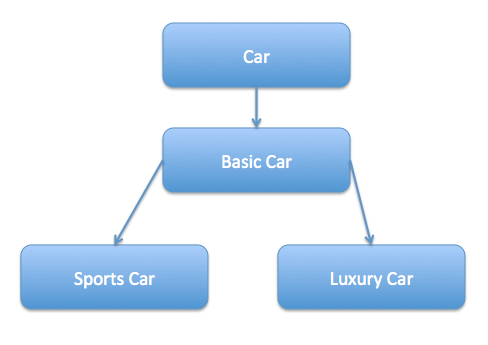
Obj

“Hello ! World”

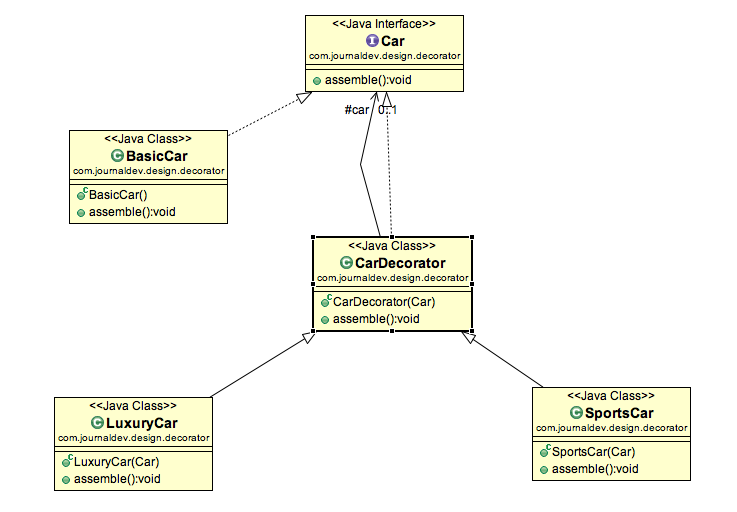
Speak()

* Decorators have the same supertype as the object they decorate
* You can use one or more decorators to wrap the object
* Given that the decorator has same supertype as the object it decorates, we can pass around a decorated object in place of the original object
* The decorator adds its own behaviour either before and/or after delegating to the object it decorates to do the rest of the job.
* Objects can be decorated at any time, so we can decorate objects dynamically at run time with as many decorators we like.

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.



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 furthermore 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.



## Implementation in JDK:

* All subclasses of java.io.InputStream, OutputStream, Reader and Writer have constructors that accept objects of their own type.
* java.util.Collections, methods checkedXXX(), synchronizedXXX() and unmodifiableXXX().
* javax.servlet.http.HttpServletRequestWrapper and HttpServletResponseWrapper

## Important Points to remember :

* 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).