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.

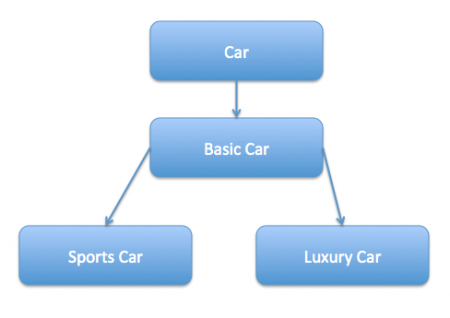
This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

We 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.

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.



/\*

\* Component interface

\* the interface or abstract class defining the methods that will be implemented.

\* In our case Car will be the component interface.

\*/

interface Car{

public void assemble();

}

/\*

\* Component Implementation

\* the basic implementation of the component interface. we can have BasicCar class as our component implementation

\*

\*/

class BasicCar implements Car{

public void assemble() {

System.out.println("Basic car");

}

}

/\*

\* Decorator

\* Decorator class implements the component interface and it has a HAS A relationship with the component interface.

\* The component variable should be accessibe to the child decorator classes, so we will make this variable protected

\*/

class CarDecorator implements Car{

protected Car car;

public CarDecorator(Car car) {

this.car = car;

}

@Override

public void assemble() {

this.car.assemble();

}

}

/\*

\* Concrete decorators

\* Extending the base decorator functionality and modifying the component behavior accordingly.

\* we can have concrete decorator classes as Luxury car and Sports car

\*/

class SportsCar extends CarDecorator{

public SportsCar(Car c) {

super(c);

}

@Override

public void assemble() {

// TODO Auto-generated method stub

super.assemble();

System.out.println("adding features of sports car");

}

}

class LuxuryCar extends CarDecorator{

public LuxuryCar(Car c) {

super(c);

}

@Override

public void assemble() {

super.assemble();

System.out.println("adding features of luxury car");

}

}

public class TestDecorator {

public static void main(String[] args) {

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

sportsCar.assemble();

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

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

sportsLuxuryCar.assemble();

}

}