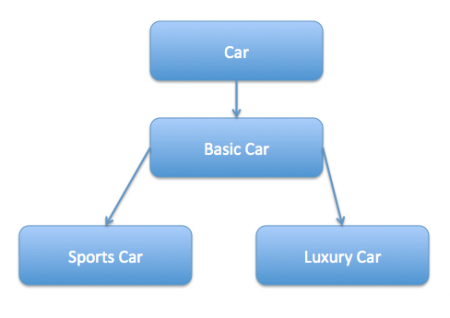
**Decorator Design Pattern**

**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.

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**](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.

**Component Implementation** – The basic implementation of the component interface. We can have BasicCar class as our component implementation.

**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.

**Concrete Decorators** – Extending the base decorator functionality and modifying the component behavior accordingly. We can have concrete decorator classes as LuxuryCar and SportsCar.

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.

