Decorator is just a linked list. We delegate the work after our decorator's work is done.

Circle is just an object of type Shape.

```java
public class Test {
    Run | Debug
    public static void main(String[] args) {
        Circle circle = new Circle();
        borderDecorator addBorder = new borderDecorator(circle);
        fillDecorator fillShape = new fillDecorator(addBorder);
        ThreeDDecorator threeD = new ThreeDDecorator(fillShape);
        threeD.draw();
    }
}
```

```java
public class Circle extends Shape {

    @Override
    public void draw() {
        System.out.println(x: "drawn a circle");
    }
}
```

As we can see, we are setting our nextShape through constructor.

Decorators :-

Therefore 3D → Fill → border → Circle.

As we are setting our next shape, and we have the decorator classes derived through Shape Decorator class, we must set our next shape through the constructor of ShapeDecorator class.

** The Shape Decorator class is derived from Shape class and contains shape class reference nextshape inside this Shape Decorator class, which makes it a possible linked list.

The Shape Decorator class extends Shape class so that it can decorate objects of type Shape.

In the draw function we call the draw function of the nextshape

```java
public abstract class Shape {
    void draw() {
        System.out.println(x: "drawing a Shape");
    }
}
```

```java
public class ShapeDecorator extends Shape {
    Shape nextShape;
    ShapeDecorator(Shape nextShape) {
        this.nextShape = nextShape;
    }

    @Override
    public void draw() {
        nextShape.draw();
    }
}
```

```java
public class borderDecorator extends ShapeDecorator {

    borderDecorator(Shape nextshape) {
        super(nextshape);
    }

    @Override
    public void draw() {
        System.out.println(x: "Adding a border to the shape....");
        super.draw(); //here ShapeDeco.draw where nextshape.draw is called
    }

}
```

```java
public class fillDecorator extends ShapeDecorator {

    fillDecorator(Shape shape) {
        super(shape);
    }

    @Override
    public void draw() {
        System.out.println(x: "Filling the shape ...");
        super.draw();
    }

}
```

next

```java
public class ThreeDDecorator extends ShapeDecorator  {

    ThreeDDecorator(Shape shape) {
        super(shape);
    }

    @Override
    public void draw() {
        System.out.println(x: "Making the shape 3D");
        super.draw();
    }
}
```

Now we create different types of decorator classes extending ShapeDecorator class.

We pass the nextshape through super keyword and set the nextShape inside ShapeDecorator parent class.

```java
public abstract class Subject {
    private List<Observer> observers;

    public Subject() {
        this.observers = new ArrayList<Observer>();
    }

    public void register(Observer o){
        observers.add(o);
    }

    public void unRegister(Observer o) {
        observers.remove(o);
    }

    public void notifyObservers() {
        for(Observer o:observers) {
            o.update(this);
        }
    }
}
```

Observer Design Pattern :-

1) Subject class :- Contains all the functionality to register, unregister and notify all observers.

- register() : Registers an observer in a collection
- unregister() : Unregisters an observer in a collection.
- NotifyObservers() : Notifies all the observer classes of the changes in data states through calling the update function of the registered observer.

```java
public interface Observer {
    void update(Subject s);
}
```

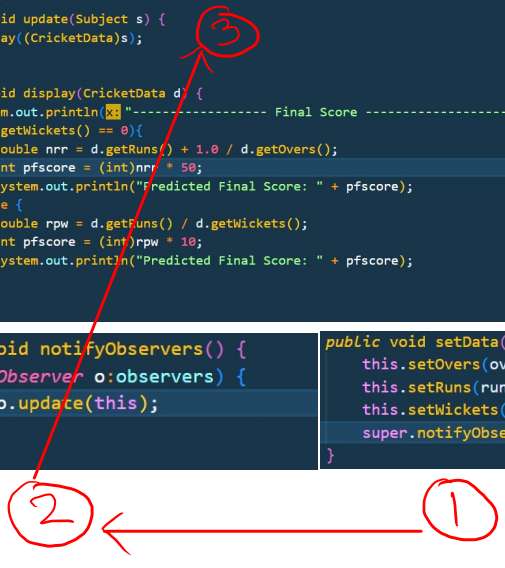
2) Observer interface is created for all the classes who potentially want to observe any data class extending the Subject class.

```java
public class FinalScorePredictionDisplay implements Observer {
    @Override
    public void update(Subject s) {
        display((CricketData)s);
    }

    public void display(CricketData d) {
        System.out.println(x: "----------------- Final Score ----------------------");
        if(d.getWickets() == 0){
            double nrr = d.getRuns() + 1.0 / d.getOvers();
            int pfscore = (int)nrr * 50;
            System.out.println("Predicted Final Score: " + pfscore);
        } else {
            double rpw = d.getRuns() / d.getWickets();
            int pfscore = (int)rpw * 10;
            System.out.println("Predicted Final Score: " + pfscore);
        }
    }
}
```

```java
public void notifyObservers() {
    for(Observer o:observers) {
        o.update(this);
    }
}
```

```java
public void setData(int runs , int wickets , int overs) {
    this.setOvers(overs);
    this.setRuns(runs);
    this.setWickets(wickets);
    super.notifyObservers();
}
```



Update function in any observer implementing class handles the Subject object, converts it into the data class, and updates the display.

This update function is only called when notifyObserverfunction is called through subject class, which only gets triggered when there is a change in the Data class.

```java
public class CricketData extends Subject {
    private int runs;

    public int getRuns() {
        return runs;
    }

    private void setRuns(int runs) {
        this.runs = runs;
    }

    private int overs;

    public int getOvers() {
        return overs;
    }

    private void setOvers(int overs) {
        this.overs = overs;
    }

    private int wickets;

    public int getWickets() {
        return this.wickets;
    }

    private void setWickets(int wickets) {
        this.wickets = wickets;
    }
```

```java
    public void setData(int runs , int wickets , int overs) {
        this.setOvers(overs);
        this.setRuns(runs);
        this.setWickets(wickets);
        super.notifyObservers();
    }
}
```

2) Subject :- Data class

The data class, being the subject extends the Subject class and contains the main data and the getter setters for these fields

```java
public class ScoreCardDisplay implements Observer {
    @Override
    public void update(Subject s) {
        display((CricketData)s);
    }

    public void display(CricketData d) {
        System.out.println(x: "------------------ Score Card ----------------------");
        System.out.println("Runs - " + d.getRuns());
        System.out.println("Wickets - " + d.getWickets());
        System.out.println("Overs - " + d.getOvers());
    }
}
```

```java
public class NetRunRateDisplay implements Observer {
    @Override
    public void update(Subject s) {
        display((CricketData)s);
    }

    public void display(CricketData d) {
        System.out.println(x: "------------------ NET Run Rate ----------------------");
        double nrr = d.getRuns() + 1.0 / d.getOvers();
        System.out.println("NRR - " + nrr);
    }
}
```

Other observer extending classes....

The test class.

```java
public class Test {
    Run | Debug
    public static void main(String[] args) {
        CricketData cd = new CricketData();
        cd.setData(runs: 2, wickets: 10, overs: 0);

        ScoreCardDisplay scDisplay = new ScoreCardDisplay();
        NetRunRateDisplay nrrDisplay = new NetRunRateDisplay();
        FinalScorePredictionDisplay fspDisplay = new FinalScorePredictionDisplay();

        cd.register(scDisplay);
        cd.register(nrrDisplay);
        cd.register(fspDisplay);

        cd.setData(runs: 99, wickets: 1, overs: 14);
        cd.setData(runs: 150, wickets: 3, overs: 20);

        cd.unRegister(fspDisplay);

        cd.setData(runs: 200, wickets: 6, overs: 30);
    }
}
```