JAVA ACADEMY - XIDERAL MONTERREY, N.L

EXERCISE - WEEK 3 DECORATOR PATTERN

Made by:
Luis Miguel Sánchez Flores

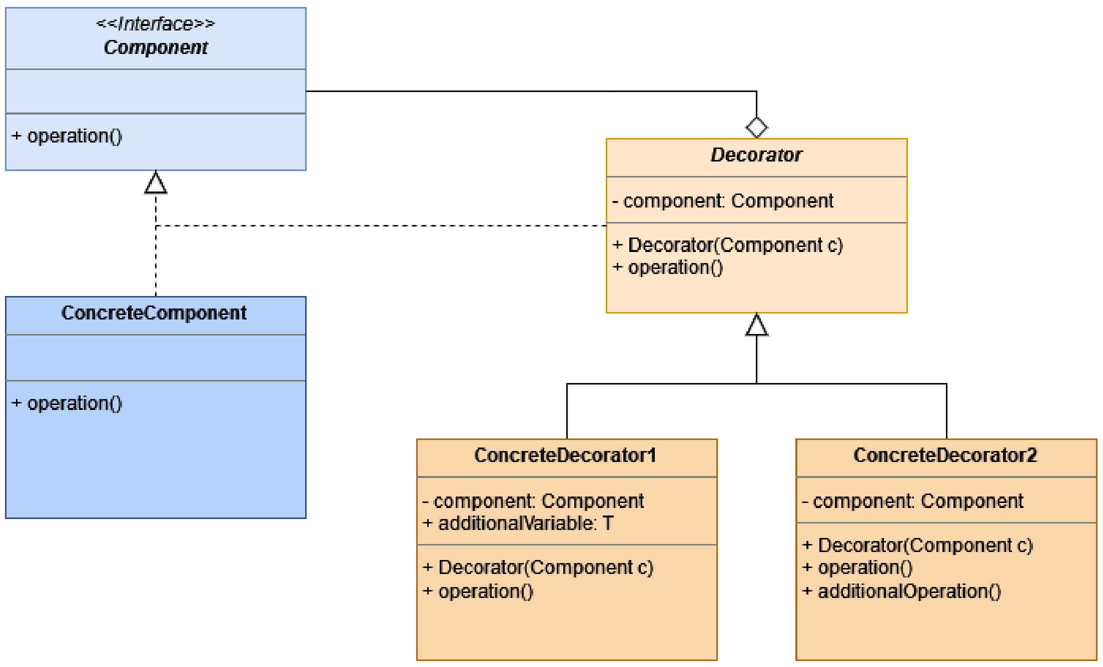
August 31st, 2024

INTRODUCTION:

The Decorator design pattern is a structural pattern that allows an object to add or override behavior in another object at runtime. Its goal is to provide a flexible way to extend the behavior of an object without modifying its underlying structure.
Some of the advantages of the Decorator pattern are its high-flexible structure enabling an easier adaptation to changing requirements, promotes code reusability through the use of multiple decorators on the same component and adheres to the Open/Close Principle (open for extension, closed for modification).
The Decorator can be considered as an alternative to developing subclasses, which can complement a class without allowing adjustments to be made at runtime.

The classes developed for the Decorator pattern are:

- **Component:** Establishes the common interface for both wrappers and wrapped objects.
- **Concrete Component:** Base object that can be wrapped for additional responsibilities / actions.
- **Decorator:** Contains a reference to a concrete componente, either a wrapped object or another decorator. The decorator defines an interface that conforms to the Component's interface.
- **Concrete Decorator:** In charge of extending the functionality / actions of the component by adding new behaviors and states.



The decorator pattern is applied when there is a need to dynamically add as well as remove responsibilities to a class, and when subclassing would be impossible due to the large number of subclasses that could result.

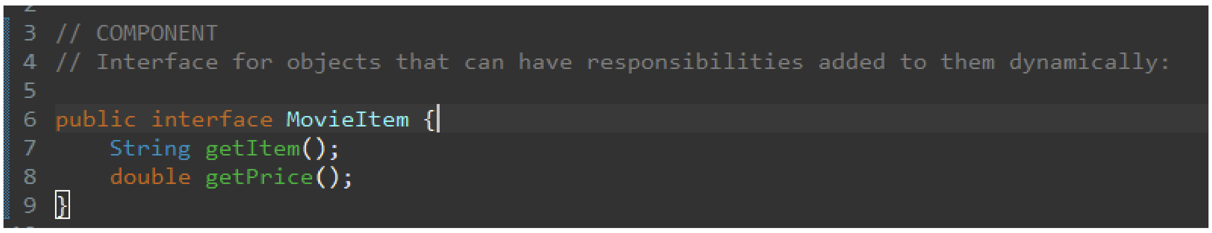
Imagine that a cinema owner wants to enhance the experience of his customers from the moment they go to buy a ticket: In addition to offering affordable tickets, the owner wants to offer a set of additional products/services that customers can buy when purchasing their tickets. The owner started proposing different options that the customer could acquire from the beginning, like a premium seating option for the customer to enjoy the movie more comfortably, access to special events such as premieres or Q&A sessions for memorable moments or simply the ability to redeem discounts to build customer loyalty, all in order to provide a personalized and unparalleled experience for each customer, and gain a competitive advantage over other local theaters.

The previous case can be significantly benefited by the use of the **Decorator** pattern, allowing the extension of a basic element such as the movie ticket to incorporate additional premium services that can further enhance the customer's movie-going experience to their liking.

JAVA EXAMPLE

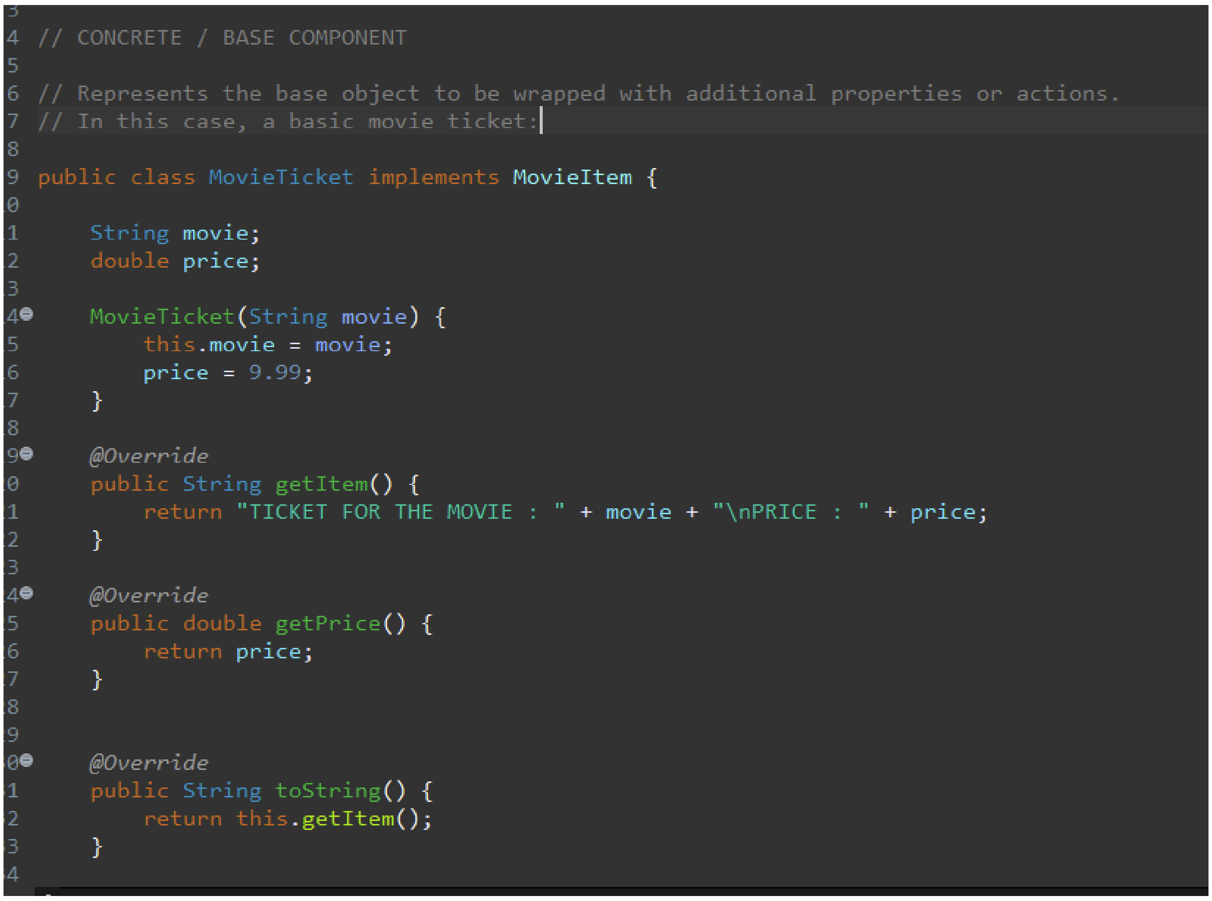
The Decorator pattern has been implemented in the following Java example:

First, the Component interface was defined for both the item to be wrapped and the set of decorators. In this case, the **MovieItem** interface sets up the abstract methods to retrieve the item's information as well as its price:



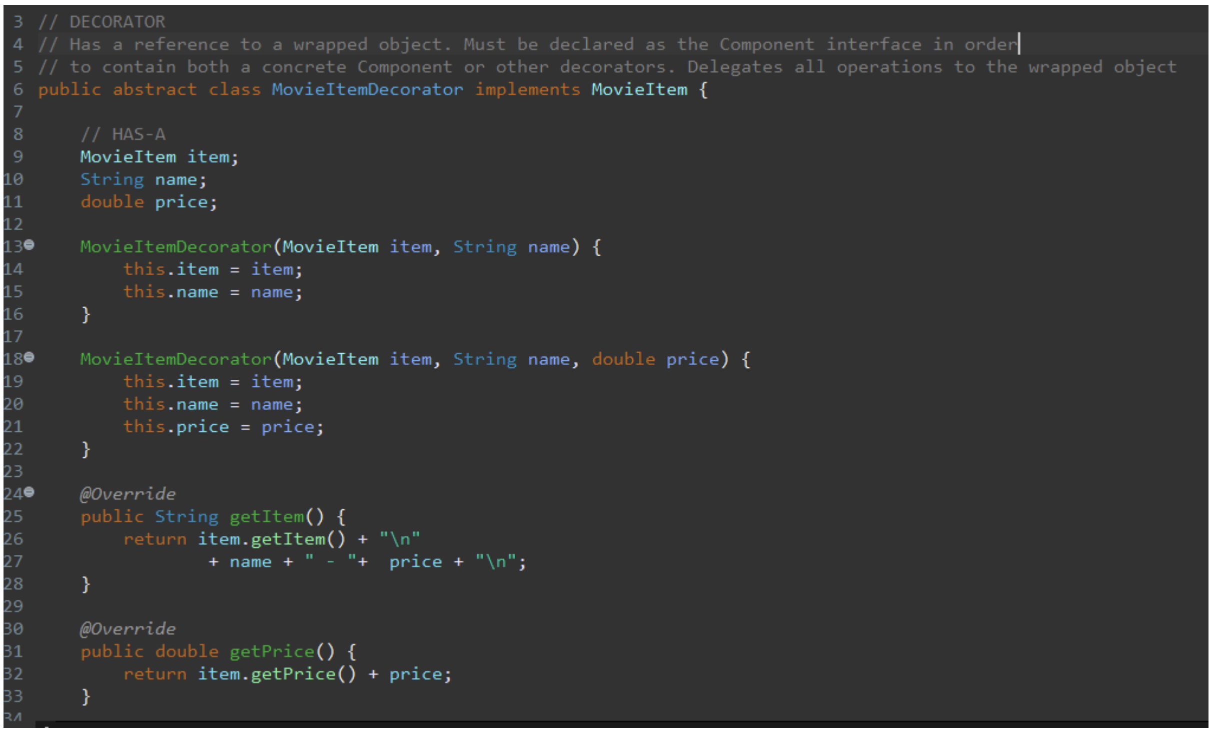
```java
// COMPONENT
// Interface for objects that can have responsibilities added to them dynamically:

public interface MovieItem {
    String getItem();
    double getPrice();
}
```

The concrete Component is created through the **MovieTicket** class that will represent the basic object in which additional properties can be added (in this case, the premium products and services).



```java
// CONCRETE / BASE COMPONENT

// Represents the base object to be wrapped with additional properties or actions.
// In this case, a basic movie ticket:

public class MovieTicket implements MovieItem {

    String movie;
    double price;

    MovieTicket(String movie) {
        this.movie = movie;
        price = 9.99;
    }

    @Override
    public String getItem() {
        return "TICKET FOR THE MOVIE : " + movie + "\nPRICE : " + price;
    }

    @Override
    public double getPrice() {
        return price;
    }


    @Override
    public String toString() {
        return this.getItem();
    }
```

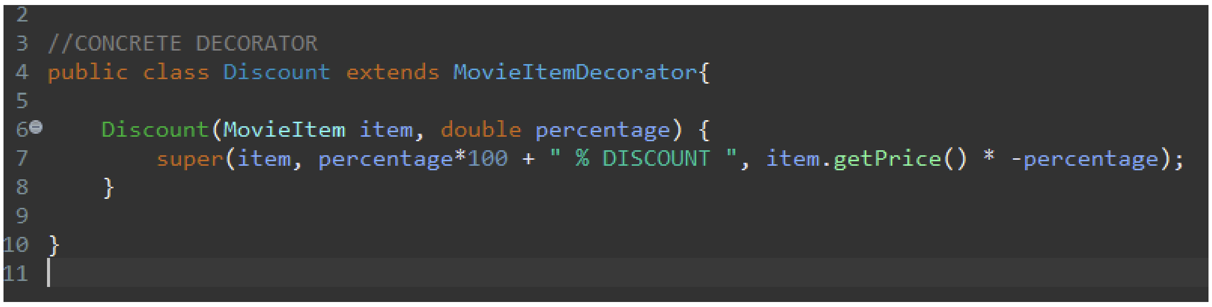
Next, the abstract Decorator class that is the **MovieItemDecorator** is established, implementing the **MovieItem** interface as well as having a reference to a specific component, whether that's the wrapped object or another decorator. It implements the method of the MovieItem interface, where it adds the item's information alongside the wrapped object's information, as well as merging the price of the wrapped object with the price of the decorator:



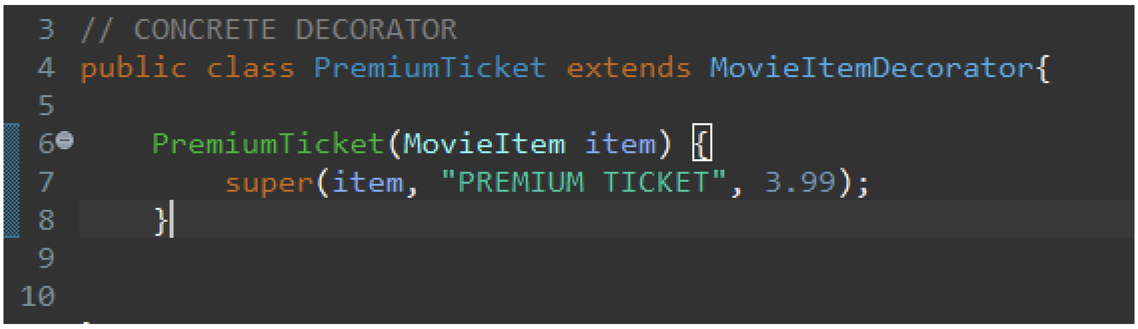
```java
// DECORATOR
// Has a reference to a wrapped object. Must be declared as the Component interface in order
// to contain both a concrete Component or other decorators. Delegates all operations to the wrapped object
public abstract class MovieItemDecorator implements MovieItem {

    // HAS-A
    MovieItem item;
    String name;
    double price;

    MovieItemDecorator(MovieItem item, String name) {
        this.item = item;
        this.name = name;
    }

    MovieItemDecorator(MovieItem item, String name, double price) {
        this.item = item;
        this.name = name;
        this.price = price;
    }

    @Override
    public String getItem() {
        return item.getItem() + "\n"
                + name + " - "+  price + "\n";
    }

    @Override
    public double getPrice() {
        return item.getPrice() + price;
    }
```

Finally, the concrete Decorators are developed to extend the functionality of the basic component that is the MovieTicket class, by adding state or adding behavior.
For this example, the Discount and PremiumTicket classes were developed to represent simple concrete decorators for the movie ticket by extending the MovieItemDecorator class. Both decorators simply call the parent constructor to define the item to be wrapped, the name of the decorator and its price.

PremimumTicket simply adds an additional charge to the price of the movie ticket, while the Discount decorator applies a reduction in the cost of said ticket, based on the provided percentage:

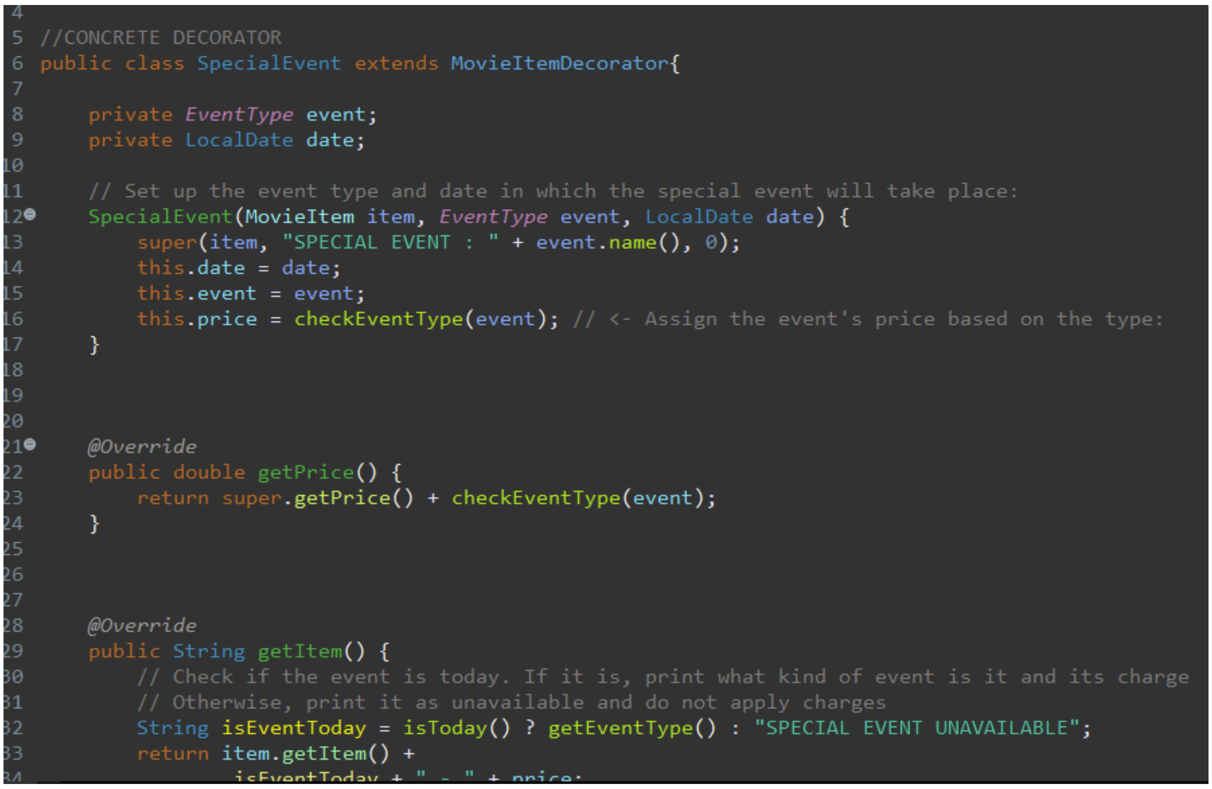


```java

//CONCRETE DECORATOR
public class Discount extends MovieItemDecorator{

    Discount(MovieItem item, double percentage) {
        super(item, percentage*100 + " % DISCOUNT ", item.getPrice() * -percentage);
    }

}

```

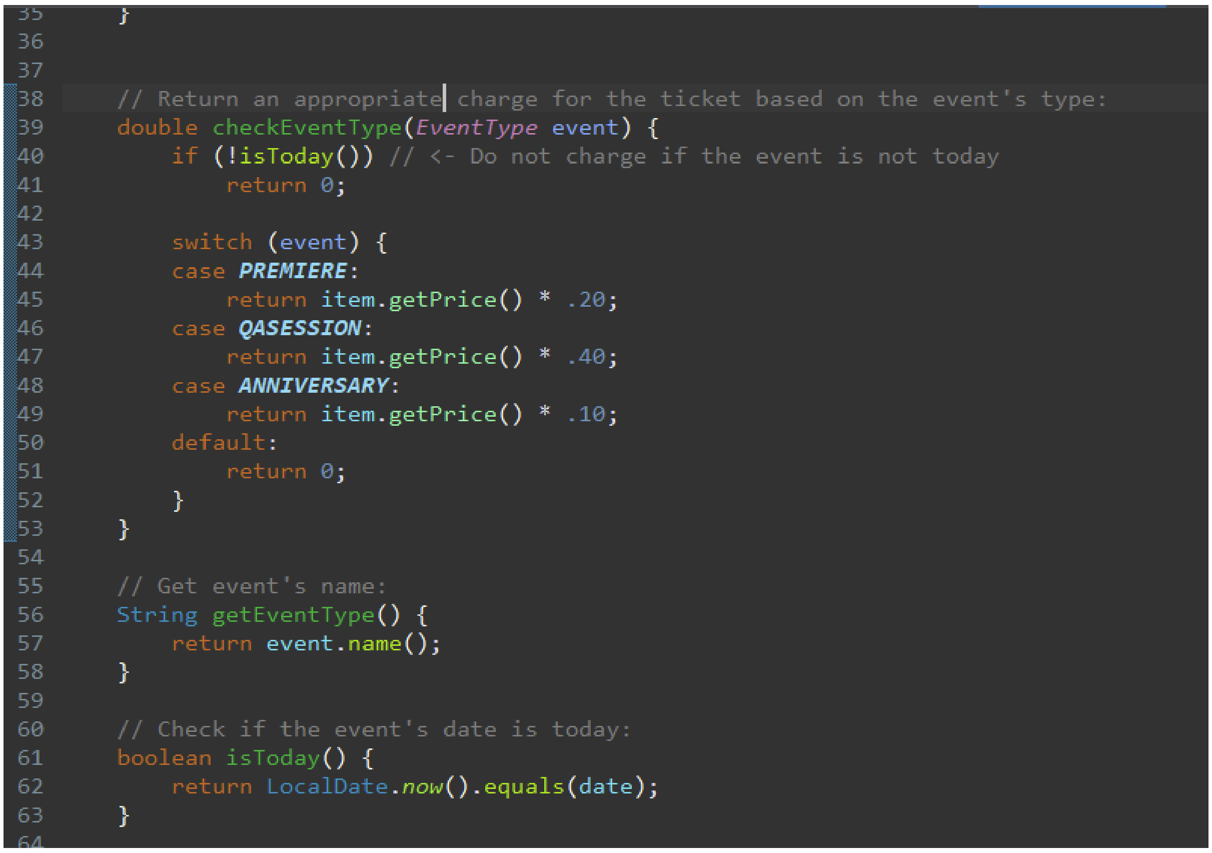


```java
// CONCRETE DECORATOR
public class PremiumTicket extends MovieItemDecorator{

    PremiumTicket(MovieItem item) {
        super(item, "PREMIUM TICKET", 3.99);
    }


```

A special concrete decorator called the **SpecialEvent** was added that overwrites the base decorator implementations to take into account what type of event the customer would attend, so that the proper charge is applied for said event:

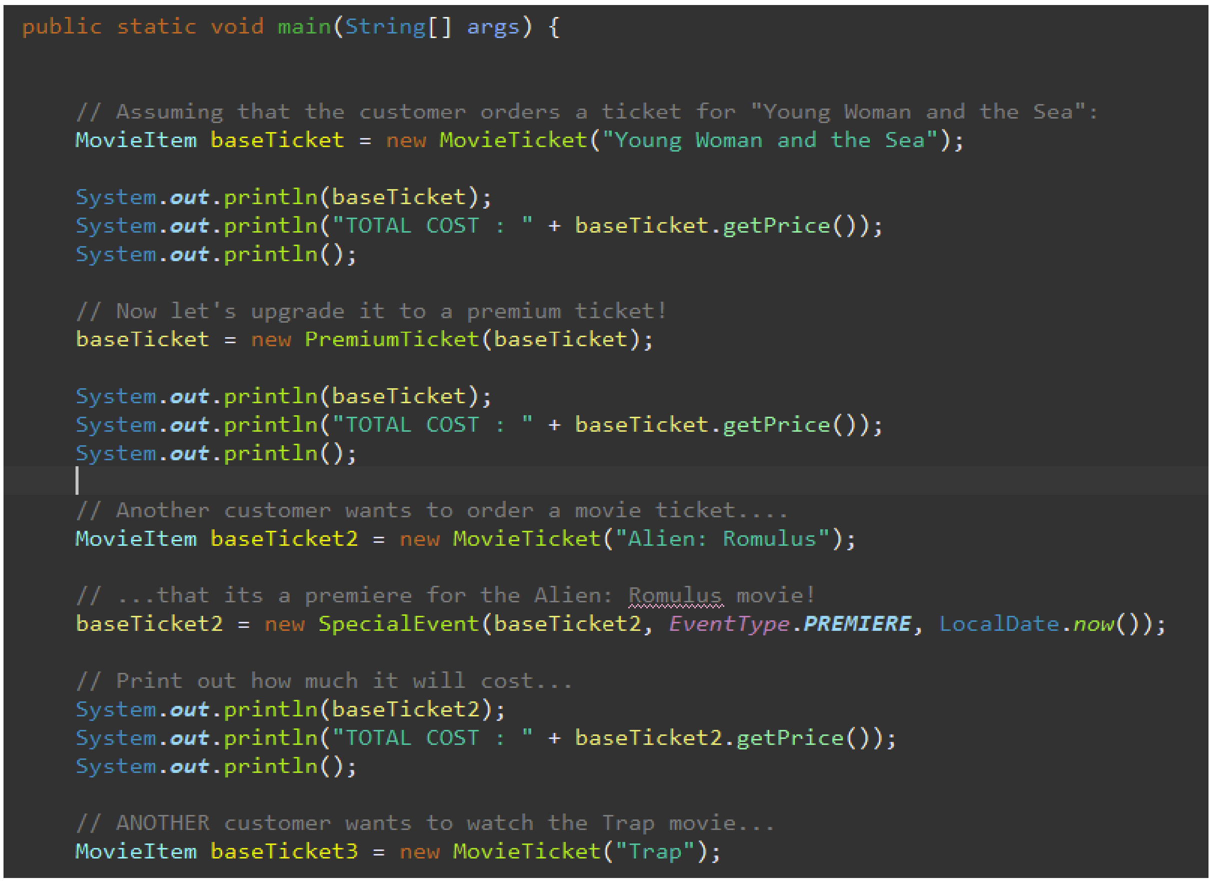


```java

//CONCRETE DECORATOR
public class SpecialEvent extends MovieItemDecorator{

    private EventType event;
    private LocalDate date;

    // Set up the event type and date in which the special event will take place:
    SpecialEvent(MovieItem item, EventType event, LocalDate date) {
        super(item, "SPECIAL EVENT : " + event.name(), 0);
        this.date = date;
        this.event = event;
        this.price = checkEventType(event); // <- Assign the event's price based on the type:
    }



    @Override
    public double getPrice() {
        return super.getPrice() + checkEventType(event);
    }



    @Override
    public String getItem() {
        // Check if the event is today. If it is, print what kind of event is it and its charge
        // Otherwise, print it as unavailable and do not apply charges
        String isEventToday = isToday() ? getEventType() : "SPECIAL EVENT UNAVAILABLE";
        return item.getItem() +
                isEventToday + " - " + price;
```

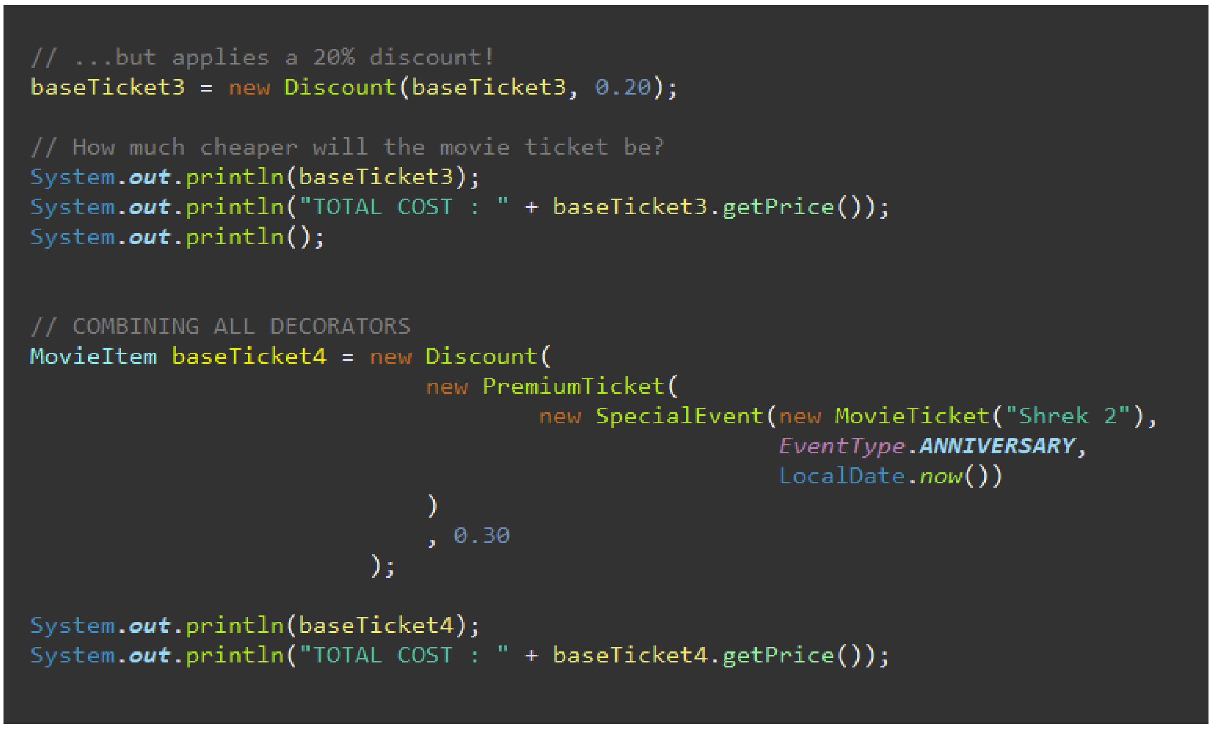


```
    }


    // Return an appropriate charge for the ticket based on the event's type:
    double checkEventType(EventType event) {
        if (!isToday()) // <- Do not charge if the event is not today
            return 0;

        switch (event) {
        case PREMIERE:
            return item.getPrice() * .20;
        case QASESSION:
            return item.getPrice() * .40;
        case ANNIVERSARY:
            return item.getPrice() * .10;
        default:
            return 0;
        }
    }

    // Get event's name:
    String getEventType() {
        return event.name();
    }

    // Check if the event's date is today:
    boolean isToday() {
        return LocalDate.now().equals(date);
    }
```

Finally, each of the concrete decorators (premium services) developed is tested with the concrete component that is the MovieTicket object, checking out how each of the decorators modifies the ticket's price:

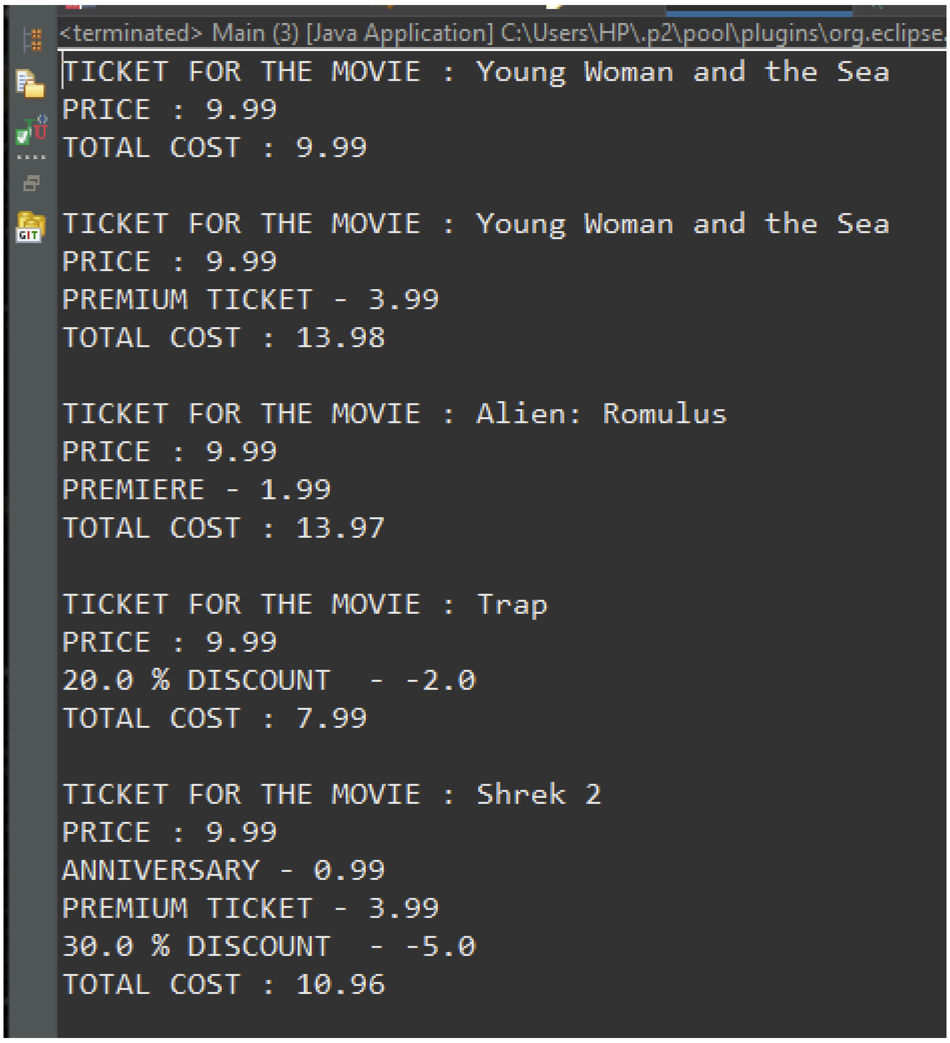


```java
public static void main(String[] args) {


    // Assuming that the customer orders a ticket for "Young Woman and the Sea":
    MovieItem baseTicket = new MovieTicket("Young Woman and the Sea");

    System.out.println(baseTicket);
    System.out.println("TOTAL COST : " + baseTicket.getPrice());
    System.out.println();

    // Now let's upgrade it to a premium ticket!
    baseTicket = new PremiumTicket(baseTicket);

    System.out.println(baseTicket);
    System.out.println("TOTAL COST : " + baseTicket.getPrice());
    System.out.println();

    // Another customer wants to order a movie ticket....
    MovieItem baseTicket2 = new MovieTicket("Alien: Romulus");

    // ...that its a premiere for the Alien: Romulus movie!
    baseTicket2 = new SpecialEvent(baseTicket2, EventType.PREMIERE, LocalDate.now());

    // Print out how much it will cost...
    System.out.println(baseTicket2);
    System.out.println("TOTAL COST : " + baseTicket2.getPrice());
    System.out.println();

    // ANOTHER customer wants to watch the Trap movie...
    MovieItem baseTicket3 = new MovieTicket("Trap");
```



```java
// ...but applies a 20% discount!
baseTicket3 = new Discount(baseTicket3, 0.20);

// How much cheaper will the movie ticket be?
System.out.println(baseTicket3);
System.out.println("TOTAL COST : " + baseTicket3.getPrice());
System.out.println();


// COMBINING ALL DECORATORS
MovieItem baseTicket4 = new Discount(
                              new PremiumTicket(
                                    new SpecialEvent(new MovieTicket("Shrek 2"),
                                                EventType.ANNIVERSARY,
                                                LocalDate.now())
                              )
                              , 0.30
                           );

System.out.println(baseTicket4);
System.out.println("TOTAL COST : " + baseTicket4.getPrice());
```

RESULT:



```
<terminated> Main (3) [Java Application] C:\Users\HP\.p2\pool\plugins\org.eclipse.
TICKET FOR THE MOVIE : Young Woman and the Sea
PRICE : 9.99
TOTAL COST : 9.99

TICKET FOR THE MOVIE : Young Woman and the Sea
PRICE : 9.99
PREMIUM TICKET - 3.99
TOTAL COST : 13.98

TICKET FOR THE MOVIE : Alien: Romulus
PRICE : 9.99
PREMIERE - 1.99
TOTAL COST : 13.97

TICKET FOR THE MOVIE : Trap
PRICE : 9.99
20.0 % DISCOUNT  - -2.0
TOTAL COST : 7.99

TICKET FOR THE MOVIE : Shrek 2
PRICE : 9.99
ANNIVERSARY - 0.99
PREMIUM TICKET - 3.99
30.0 % DISCOUNT  - -5.0
TOTAL COST : 10.96
```

With the use of the Decorator pattern, it allows us to add additional properties and functionalities to a component when necessary, as was the case with the movie ticket where desired premium services can be added to improve the customer experience from the start.