# Decorator

## Nested (Internal) Classes

- Some classes should not be accessed outside of the context of other types of objects.
  - E.g. Class `Node` in a `LinkedList` class - it does not make sense to use `Node` independently.
- Inner classes have access to the states (instance variables) of the outer class.
  - But you have to tell the compiler where to look for the said variable.
  - For example:

```
public class MyLinkedList<E> implements MyList<E> {
    private Node<E> headNode = null;

    private class MyLinkedListIterator<E> implements Iterator
        Node<E> next;

        public MyLinkedListIterator() {
          // Tells the compiler to look for headNode in MyLi
            this.next = (Node<E>) MyLinkedList.this.headNode
        }
    }
}
```

## The Decorator Pattern

- The **Decorator Design Pattern** is used to add behavior or responsibilities to an object dynamically without affecting the behavior of other objects from the same class.
- This pattern is often used for adhering to the **Single Responsibility Principle** because it allows you to divide functionality between classes with unique concerns.

- **Components:**
  - **Base Class:**
    - The **base class** is the core functionality class, which serves as the foundation or the starting point.
    - This base class can be an **abstract class or an interface (normally an interface)** that defines the contract for what behaviors or methods the components should have.
    - The **concrete class implementing this base class** will provide the default or base functionality.
  - **Decorator Classes:**
    - A **decorator class** is essentially the decorator in the Decorator Pattern.
    - The decorator class "wraps" the base class (or other wrappers) to add new functionalities.
    - The decorator class has a reference to the base class (or any subclass or interface of it), and it forwards requests to the wrapped object.
    - However, it can also extend or alter the behavior of the base class by adding extra logic before or after calling the base class's methods.
    - **Anatomy of a Decorator Class:**
      - A decorator class encapsulates an instance of the same interface that it implements.
      - The constructor takes an instance of the interface as a parameter.
      - Instead of implementing functionality directly, methods delegate to the encapsulated instance.
    - **Decorator objects are typically used and preferred over the base object.**
    - **It is NOT possible to create decorator instances without first creating a base instance.**
  - **Diagram**:

```
        +--------------------+
        |  Base Interface    |
        +--------------------+ <------------------+ <-----
        |  + operation()     |                    |
        +--------------------+                    |
                  ^                               |
                  |                               |
        +---------------------+          +----------------
        | Base Concrete Class |          |    Decorator A
        +---------------------+          +----------------
        | + operation()       |          | + operation()
        +---------------------+          | + extraBehavio
                                         +----------------
                                                (more dec
```

- **Decorator Pattern “Recipe”:**
  - **Implement**: Make a new class that implements the base interface.
  - **Encapsulate**: Encapsulate another instance of the interface inside the new class.
  - **Delegate**: Forward (delegate) all methods to the other instance.
  - **Modify**: Selectively add or change method functionality as desired.
- **Example**:
  - **Base Interface:**

```
public interface PriceTag {
    void setAmount(double amount);
    double getAmount();
}
```

  - **Base Class - Implementation:**

```
public class PriceTagImpl implements PriceTag {
    private double amount;

    PriceTagImpl(double amount) {
        this.amount = amount;
    }

    @Override
    public void setAmount(double amount) {
        this.amount = amount;
    }

    @Override
    public double getAmount() {
        return amount;
    }
}
```

- **Decorated Price Tag:**

```
public class DiscountedPriceTag implements PriceTag {
    private PriceTag tag; // Encapsulate the base interfac
    private double discount;

    // Takes in an instance of the base interface
    DiscountedPriceTag(PriceTag tag, double discount) {
        this.tag = tag;
        this.discount = discount;
    }

    @Override
    public void setAmount(double amount) {
        tag.setAmount(amount);
    }
```

```
    @Override
    public double getAmount() {
      // Instead of re-implementing the PriceTag behavior,
      // delegates to the encapsulated PriceTag object (ge
        return Math.max(tag.getAmount() - discount, 0);
    }
}
```

- **Chaining Multiple Decorator:**
  - **Chaining decorators** means wrapping one decorator around another, creating layers of decoration.
  - Each decorator adds behavior while still delegating the core operation to the next object in the chain (which could be the original component or another decorator).
  - Example:

```
PriceTag basicTag = new PriceTagImpl(100);
PriceTag discountTag1 = new DiscountedPriceTag(basicTag, 2
PriceTag discountTag2 = new DiscountedPriceTag(discountTag
discountTag2.getAmount();
```

  - Or:

```
// ConcreteComponent
Coffee simpleCoffee = new SimpleCoffee();

// Add Milk Decorator
Coffee milkCoffee = new MilkDecorator(simpleCoffee);

// Add Sugar Decorator on top of Milk Decorator
Coffee milkSugarCoffee = new SugarDecorator(milkCoffee);

// Now we have a coffee with milk and sugar
System.out.println(milkSugarCoffee.getDescription() + " $"
```

- **Unwrapping Decorators:**
  - Unwrapping decorators involves accessing the original object or any intermediate object in the chain of decorators.
  - This is useful if you want to perform some operation on the base component or need to retrieve the original object for some reason.
  - **Limitations**:
    - The unwrapping process assumes that decorators are carefully stacked and that you're aware of their types at runtime (i.e., using `instanceof` checks).
    - Overuse of unwrapping can complicate your code, so it's best used sparingly and when necessary.

```
// Modify the CoffeeDecorator (base concrete class) to allow
public abstract class CoffeeDecorator implements Coffee {
    protected Coffee decoratedCoffee;

    public CoffeeDecorator(Coffee coffee) {
        this.decoratedCoffee = coffee;
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription();
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost();
    }

    // New method to access the wrapped object (can be named
    public Coffee getDecoratedCoffee() {
        return decoratedCoffee;
```

```
    }
}

// Example usage:
public class CoffeeShop {
    public static void main(String[] args) {
        Coffee simpleCoffee = new SimpleCoffee();
        Coffee milkCoffee = new MilkDecorator(simpleCoffee);
        Coffee milkSugarCoffee = new SugarDecorator(milkCoffe

        System.out.println(milkSugarCoffee.getDescription() +

        // Unwrapping: Let's access the underlying milkCoffee
        if (milkSugarCoffee instanceof CoffeeDecorator) {
            Coffee unwrappedMilkCoffee = ((CoffeeDecorator) m
            System.out.println("Unwrapped: " + unwrappedMilkC
        }

        // Unwrapping further: Let's access the original simp
        if (unwrappedMilkCoffee instanceof CoffeeDecorator) {
            Coffee unwrappedSimpleCoffee = ((CoffeeDecorator)
            System.out.println("Further Unwrapped: " + unwrap
        }
    }
}

// Output:
// Simple Coffee, Milk, Sugar $7.0  (Type: SugarDecorator)
// Unwrapped: Simple Coffee, Milk $6.5 (Type: MilkDecorator)
// Further Unwrapped: Simple Coffee $5.0 (Type: SimpleCoffee)

// Explanation:
// Chaining: We created milkCoffee and then wrapped it with S
// to get milkSugarCoffee. Both decorators add their own beha
// underlying object.
// Unwrapping: We can access each decorator and the base obje
```

```
// through the getDecoratedCoffee() method. By repeatedly unw
// you can get back to the original object in the chain.
```

- **Limitations of the Decorator Pattern:**

1. **Multiple Decorations Must Be Managed by the Programmer**

    - When decorators are applied, the programmer is responsible for ensuring the correct sequence of decorations, handling compatibility issues between different decorators, and avoiding multiple redundant decorations. Here are some challenges:

    a) **Does Order Matter?**

    - Yes, the order in which decorators are applied **can matter**. Since each decorator adds or modifies behavior before or after delegating to the next object in the chain, the final output may differ depending on the order of decorations.
    - For example:

      ```
      Coffee coffeeWithMilkThenSugar = new SugarDecorator
      (new MilkDecorator(new SimpleCoffee()));
      Coffee coffeeWithSugarThenMilk = new MilkDecorator(n
      ew SugarDecorator(new SimpleCoffee()));
      ```

      ⇒ Both combinations of decorators would give different intermediate descriptions:

        - `coffeeWithMilkThenSugar`: "Simple Coffee, Milk, Sugar"
        - `coffeeWithSugarThenMilk`: "Simple Coffee, Sugar, Milk"

    b) **Are Some Decorations Incompatible with Each Other?**

    - Yes, some decorators might be incompatible. For example:
        - **Mutually exclusive behaviors**: Imagine a coffee decorator that adds "extra milk" and another decorator that adds "no milk." Applying both would create a conflict.

        - **Incorrect combinations**: Some decorators may assume certain conditions. If one decorator expects a specific state (like a non-null field or certain initialization) but another decorator violates that condition, it could lead to runtime errors.
    - Managing such cases is the programmer's responsibility, requiring extra vigilance when applying multiple decorators. Some solutions could involve:
        - Implementing checks within the decorator constructors to prevent incompatible combinations.
        - Using validation or composition rules to enforce correct decoration combinations.

    c) **What If the Same Decoration Is Added Multiple Times?**

    - If the same decorator is applied multiple times, it may result in **duplicate behavior**, **redundant operations**, or inconsistent states. For example, applying a `MilkDecorator` twice to the same coffee object would describe the coffee as having "Milk, Milk."
    - Solutions:
        - Implementing checks within each decorator to prevent it from being applied multiple times (perhaps by keeping track of the current state of decorations).
        - Using a decorator manager or factory that ensures each decorator is only applied once.

2. **No Access to Encapsulated Object's Protected Fields**

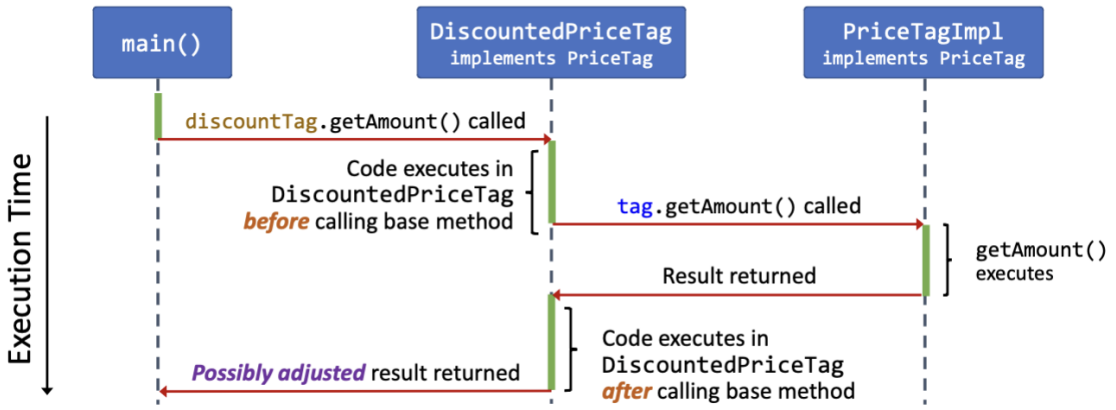
- Since decorators operate through composition, they don't have direct access to the internal (protected or private) state of the encapsulated object.

- **Tracing Method Execution:**
    - **Example:**

```
PriceTag basicTag = new PriceTagImpl(100);
PriceTag discountTag = new DiscountedPriceTag(basicTag, 2(
```

```
discountTag.getAmount();
```

- **Execution Trace:**




  - `getAmount()` in `DiscountedPriceTag` is called, which calls `getAmount()` of `PriceTag` base interface, which refers to the concrete implementation of `getAmount()` in the base concrete class, `PriceTagImpl`.
  - `getAmount()` in base concrete class `PriceTagImpl` executes and return a value to `DiscountedPriceTag`'s `getAmount()`, which may adjust or simply return that value.