Decorator Pattern

Introduction

- Decorators have the same supertype as the objects they decorate.
- You can use one or more decorators to wrap an object.
- Given that the decorator has the same supertype as the object it decorates, we can pass around a decorated object in place of the original (wrapped) object.
- The decorator adds its own behavior either before and/or after delegating to the object it decorates to do the rest of the job.
- Objects can be decorated at any time, so we can decorate objects dynamically at runtime with as many decorators as we like.

Definition: **The Decorator Pattern** attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Design Principle: Classes should be open for extension, but closed for modification.

Our goal is to allow classes to be easily extended to incorporate new behaviors without modifying existing code.

Motivation

As an example, consider a beverage in Starbucks. One may wish to add condiment **Mocha** or **Whip** to it, as appropriate. Assume beverages are represented by instances of the **Beverage** class, and assume this class has no functionality for adding condiments. One could create a subclass **BeverageWithCondiments** that provides them, or create a **CondimentDecorator** that adds this functionality to existing **Beverage** objects. At this point, either solution would be fine.

Now, assume one also desires the ability to add **Vegetables** to beverages. Again, the original **Beverage** class has no support. The **BeverageWithCondiments** subclass now poses a problem, because it has effectively created a new kind of beverage. If one wishes to add **vegetables** support to many but not all beverages, one must create subclasses **BeverageWithVegetables** and **BeverageWithVegetablesAndCondiments** etc. This problem gets worse with every new feature or beverage subtype to be added. For the decorator solution, we simply create a new **VegetablesDecorator** —at runtime, we can decorate existing beverages with the **VegetablesDecorator** or the **CondimentDecorator** or both, as we see fit. Notice that if the functionality needs to be added to all beverages, you could modify the base class and that will do. On the other hand, sometimes (e.g., using external frameworks) it is not possible, legal, or convenient to modify the base class.

Example

Starbucks Coffee

1. Take a House Blend Coffee object.
2. Decorate it with a Mocha object.
3. Decorate it with a Whip object.
4. Call the cost() method and rely on delegation to add on the condiment costs.

Beverage Abstract Class

```
public abstract class Beverage {

	String description = "Unknown Beverage";

	public String getDescription() {
		return description;
	}

	public abstract double cost();

	public String toString() {
		return getDescription() + " costs " + cost();
	}
}
```

Beverage Concrete Class

```
//	Espresso costs $1.99
public class Espresso extends Beverage {

	public Espresso() {
		this.description = "Espresso";
	}

	@Override
	public double cost() {
		return 1.99;
	}

}
```

```
//    House Blend Coffe costs $0.89
public class HouseBlend extends Beverage {

      public HouseBlend() {
            this.description = "House Blend Coffee";
      }

      @Override
      public double cost() {
            return 0.89;
      }

}
```

Condiment Decorator abstract class

```
public abstract class CondimentDecorator extends Beverage {
      public abstract String getDescription();
}
```

Condiment Decorator concrete class

```
//    Moch costs $0.20
public class Mocha extends CondimentDecorator {

      Beverage beverage;

      public Mocha(Beverage beverage) {
            this.beverage = beverage;
      }

      @Override
      public String getDescription() {
            return this.beverage.getDescription() + ", Mocha";
      }

      @Override
      public double cost() {
            return this.beverage.cost() + 0.20;
      }

}
```

```
//    Whip costs $0.30
public class Whip extends CondimentDecorator {

	Beverage beverage;

	public Whip(Beverage beverage) {
		this.beverage = beverage;
	}

	@Override
	public String getDescription() {
		return this.beverage.getDescription() + ", Whip";
	}

	@Override
	public double cost() {
		return this.beverage.cost() + 0.30;
	}

}
```

Starbucks test class

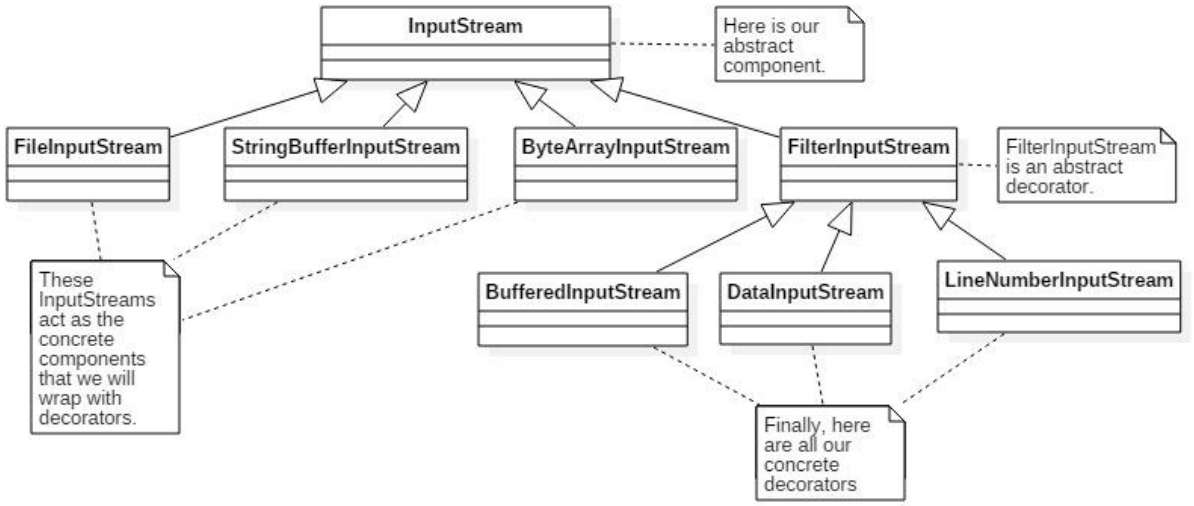
```
public class StarbucksCoffee {

	public static void main(String[] args) {
		Beverage beverage = new Espresso();
		System.out.println(beverage.toString());
		// -> Espresso costs 1.99

		Beverage beverage2 = new HouseBlend();
		beverage2 = new Mocha(beverage2);
		beverage2 = new Whip(beverage2);
		System.out.println(beverage2.toString());
		// -> House Blend Coffee, Mocha, Whip costs 1.39
	}

}
```

Example – Real World Decorator: Java I/O



Writing out own Java I/O Decorator

Write a decorator that converts all uppercase characters to lowercase in the input stream. “MmMm” -> “mmmm”

```
import java.io.*;

//Extend the FilterInputStream, the abstract decorator for all InputStreams.
public class LowerCaseInputStream extends FilterInputStream{

	public LowerCaseInputStream(InputStream in) {
		super(in);
	}

	public int read() throws IOException {
		int c = in.read();
		return (c == -1 ? c : Character.toLowerCase((char)c));
	}

	public int read(byte[] b, int offset, int len) throws IOException {
		int result = in.read(b, offset, len);
		for (int i = offset; i < offset + result; i++) {
			b[i] = (byte) Character.toLowerCase((char)b[i]);
		}
		return result;
	}

}
```

Testing class

```
import java.io.*;

public class InputTest {

	public static void main(String[] args) {
		int c;

		try {
			InputStream in = new LowerCaseInputStream(
					new BufferedInputStream(
							new FileInputStream("test.txt")));

			while ((c = in.read()) >= 0) {
				System.out.print((char)c);
			}

			in.close();
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
}
```

test.txt file content: I Know.

Output: i know.

Bullet Points

- Inheritance is one form of extension, but not necessarily the best way to achieve flexibility in our designs.
- In our designs, we should allow behavior to be extended without the need to modify existing code.
- Composition and delegation can often be used to add new behaviors at runtime.
- The Decorator Pattern provides an alternative to subclassing for extending behavior.
- The Decorator Pattern involves a set of decorator classes that are used to wrap concrete components.
- Decorator classes mirror the type of the components they decorate. (In fact, they are the same type as the components they decorate, either through inheritance or interface implementation.)
- Decorators change the behavior of their components by adding new functionality before and/or after (or even in place of) method calls to the component.
- You can wrap a component with any number of decorators.
- Decorators are typically transparent to the client of the component; that is, unless the client is relying on the component's concrete type.
- Decorators can result in many small objects in our design, and overuse can be complex.