COMP2511

Decorator Pattern
Adapter Pattern

Prepared by

Dr. Ashesh Mahidadia

Design Patterns

- Creational Patterns
  - Abstract Factory
  - Factory Method
  - Singleton
- Structural Patterns
  - Adapter *This week*
  - Composite *discussed*
  - Decorator *This week*
- Behavioral Patterns
  - Iterator *discussed*
  - Observer *discussed*
  - State *discussed*
  - Strategy *discussed*
  - Template
  - Visitor

We plan to discuss the rest of the design patterns above in the following weeks

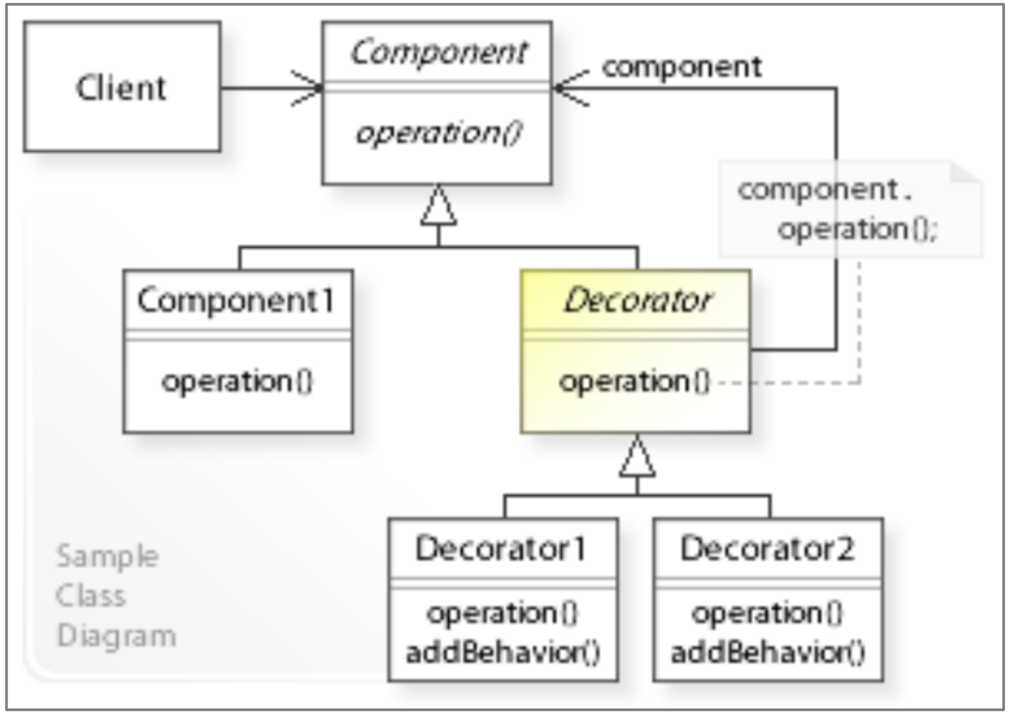
Decorator Pattern

Decorator Pattern: Intent

- "Attach additional responsibilities to an object dynamically.
  Decorators provide a flexible alternative to sub-classing for extending functionality."
  [GoF]
- Decorator design patterns allow us to selectively add functionality to an object (not the class) at runtime, based on the requirements.
- Original class is not changed (Open-Closed Principle).
- Inheritance extends behaviors at compile time, additional functionality is bound to all the instances of that class for their life time.
- The decorator design pattern prefers a composition over an inheritance.
  Its a structural pattern, which provides a wrapper to the existing class.
- Objects can be decorated multiple times, in different order, due to the recursion involved with this design pattern. See the example in the Demo.
- Do not need to implement all possible functionality in a single (complex) class.

Decorator Pattern: Structure

- *Client* : refers to the Component interface.
- *Component*: defines a common interface for *Component1* and *Decorator* objects
- *Component1* : defines objects that get decorated.
- *Decorator*: maintains a reference to a *Component* object, and forwards requests to this component object (*component.operation()*)
- *Decorator1, Decorator2, …* : Implement additional functionality (*addBehavior*()) to be performed before and/or after forwarding a request.



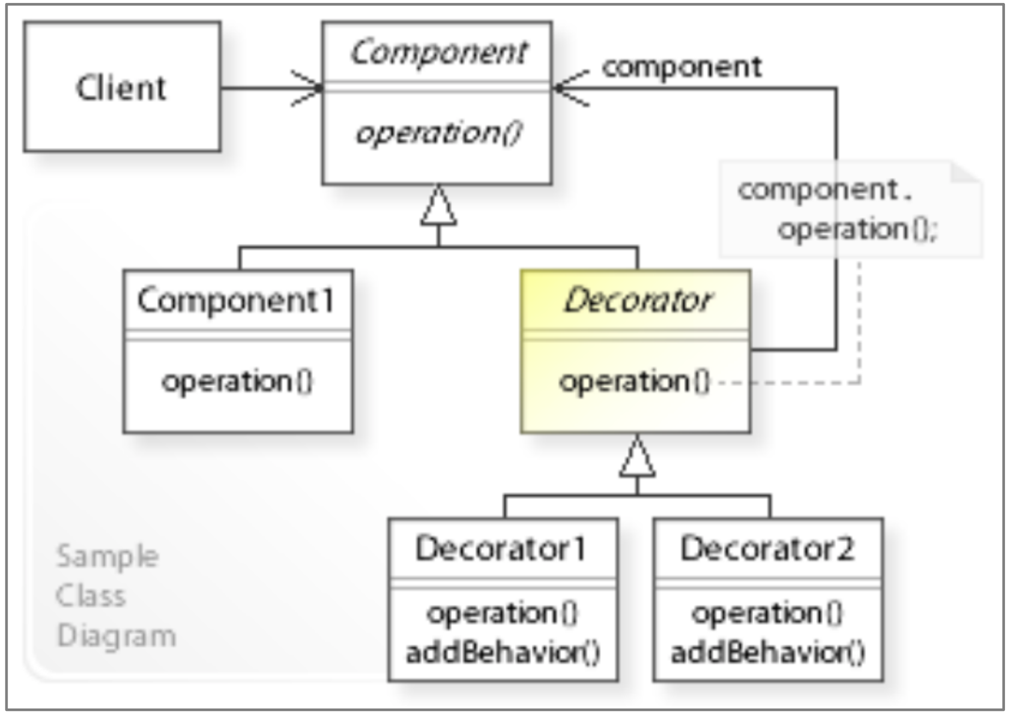
See the example in the Demo.

Decorator Pattern: Structure

- 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.

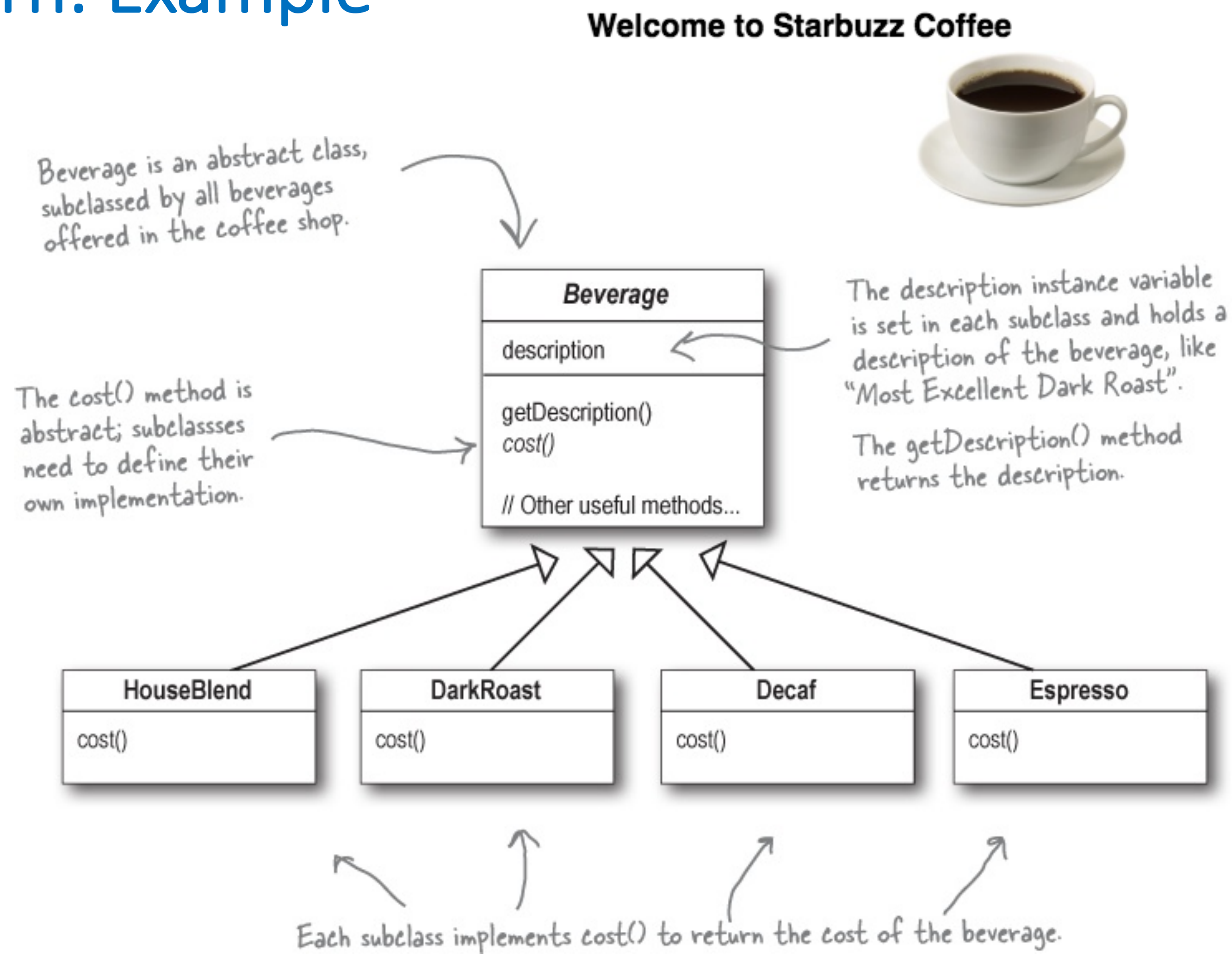
- The decorator adds its own behavior either before and/or after delegating to the object it decorates to do the rest of the job.



From the book "Head First Design Pattern".

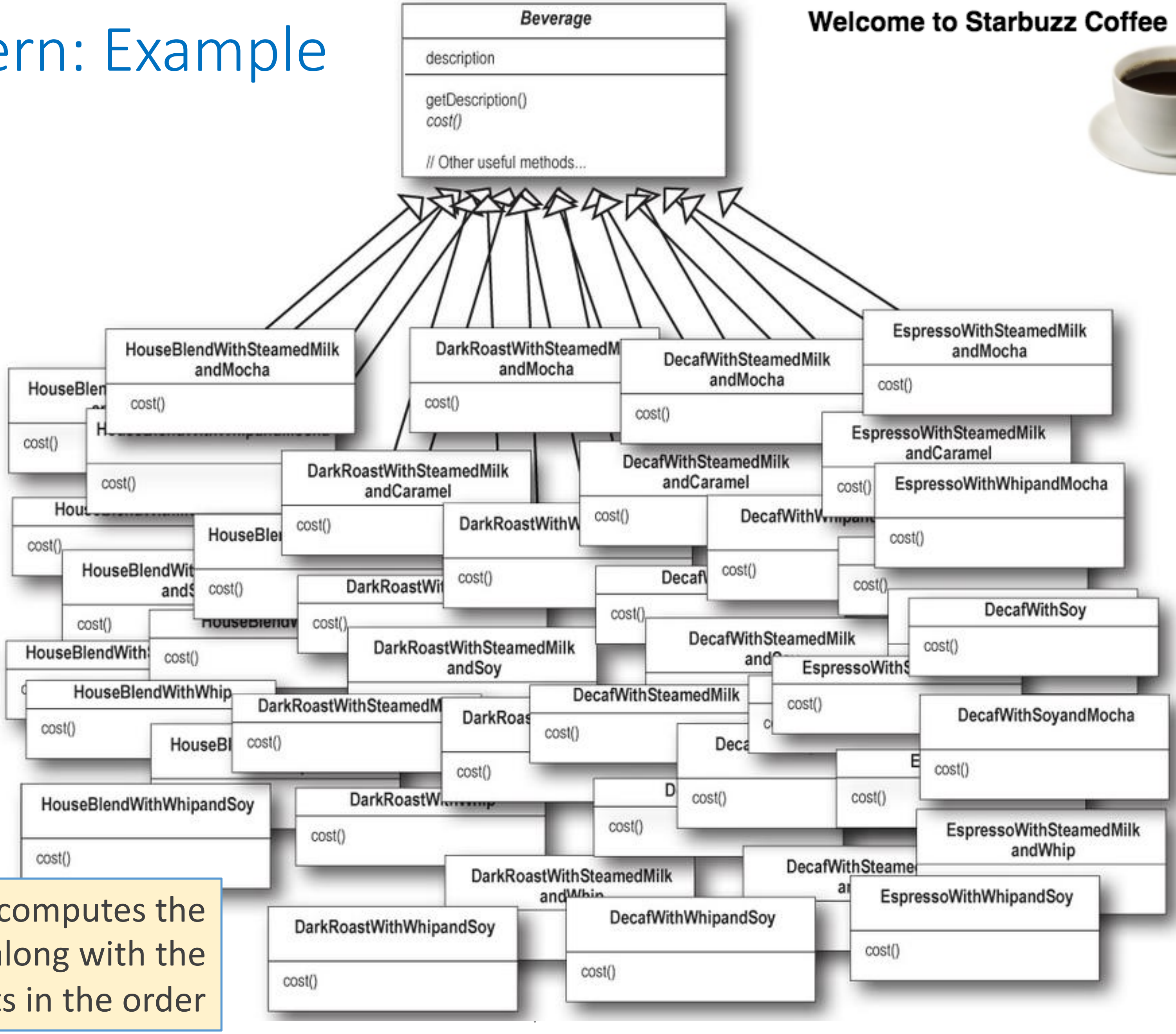
See the example in the Demo.

Decorator Pattern: Example



Decorator Pattern: Example

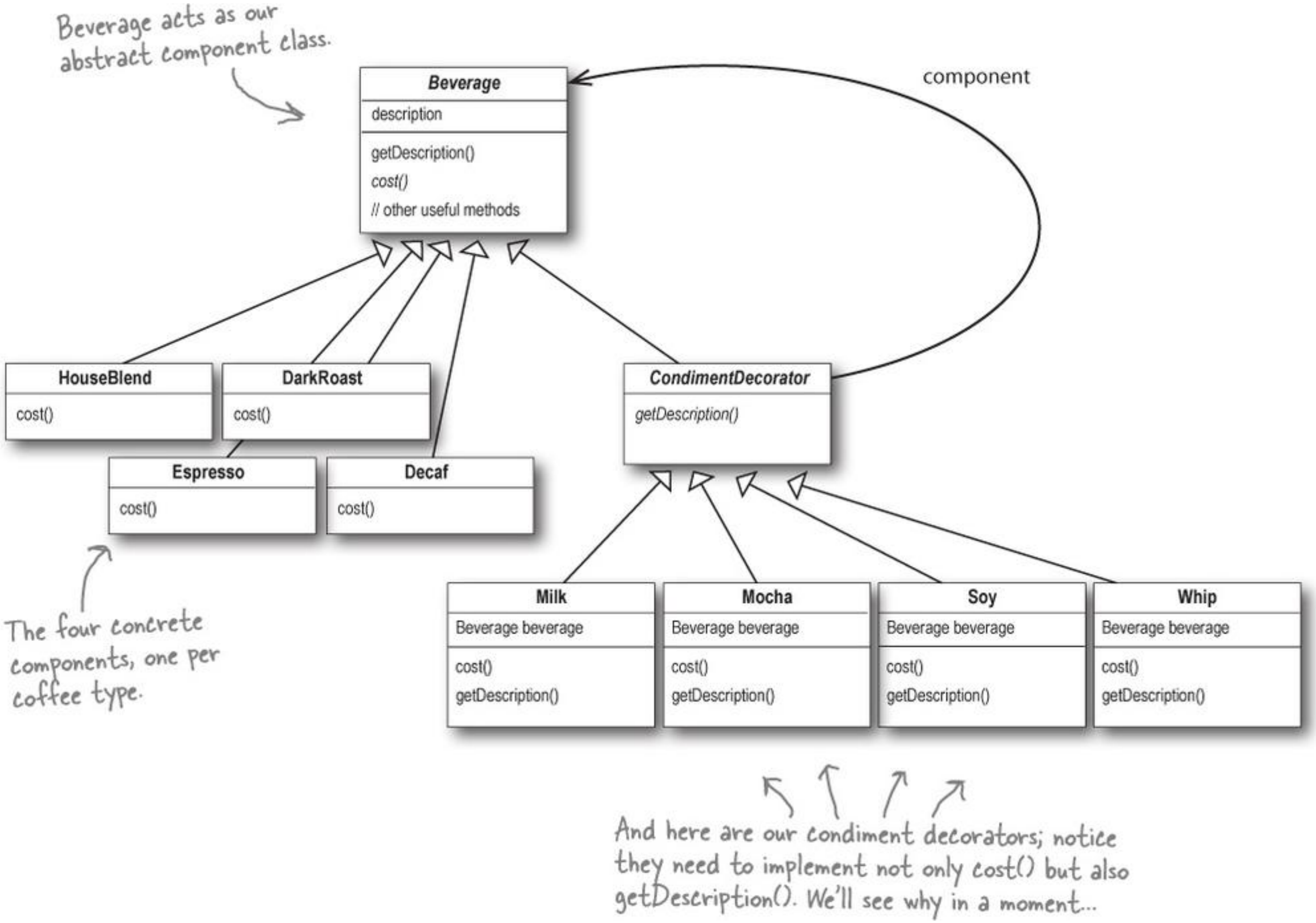
Welcome to Starbuzz Coffee



Too many combinations to consider!

Each cost method computes the cost of the coffee along with the other condiments in the order

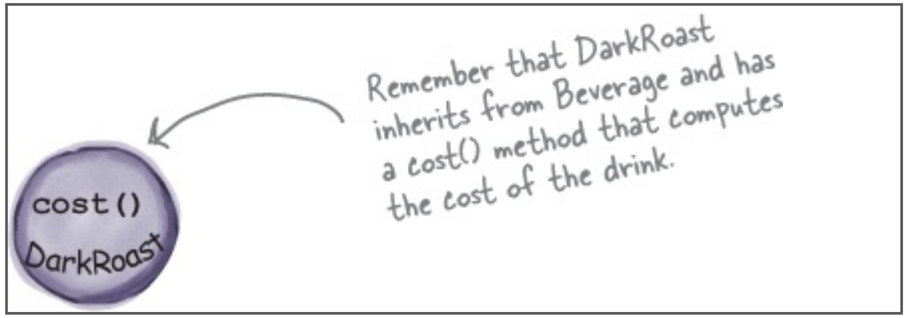
Decorator Pattern: Example

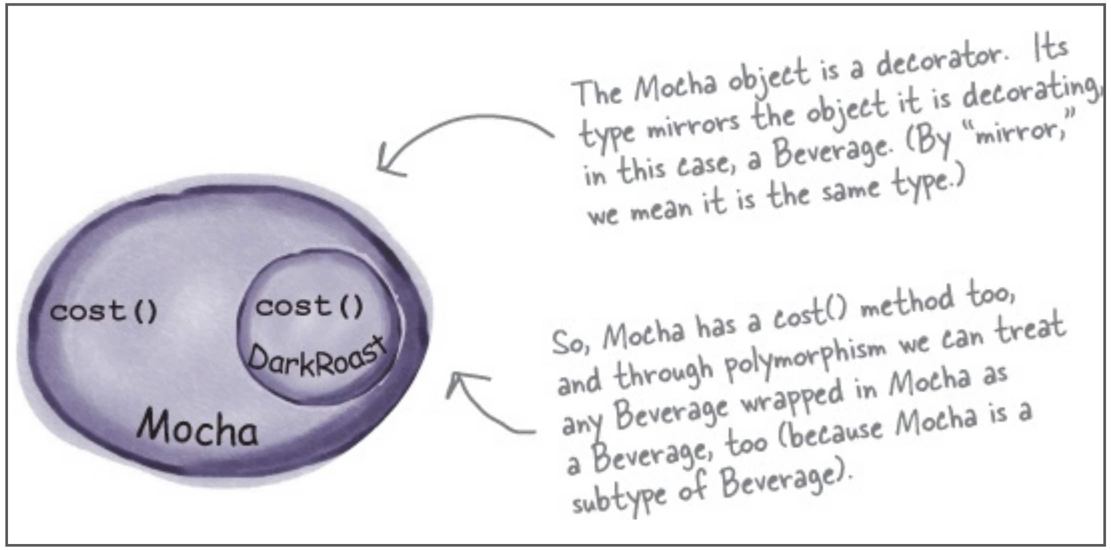


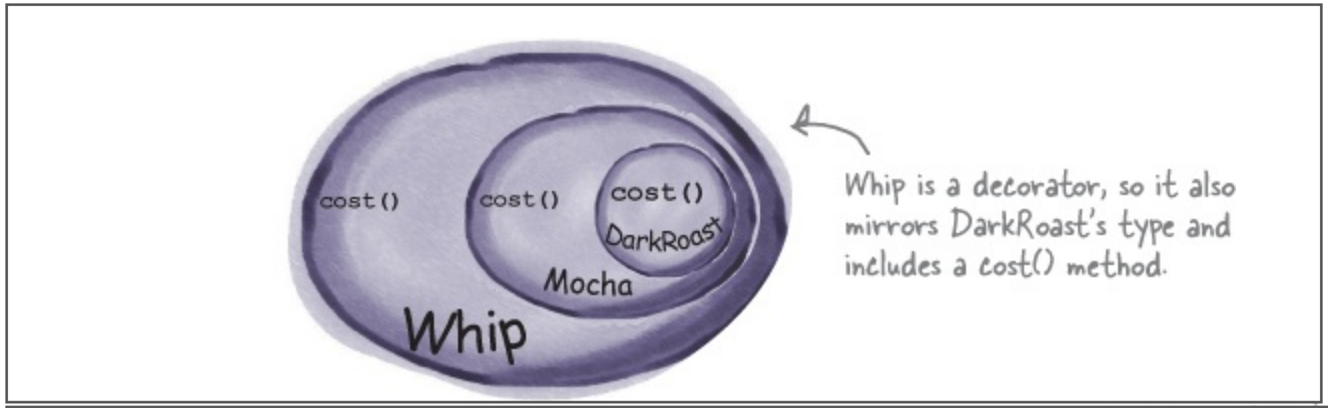
Welcome to Starbuzz Coffee

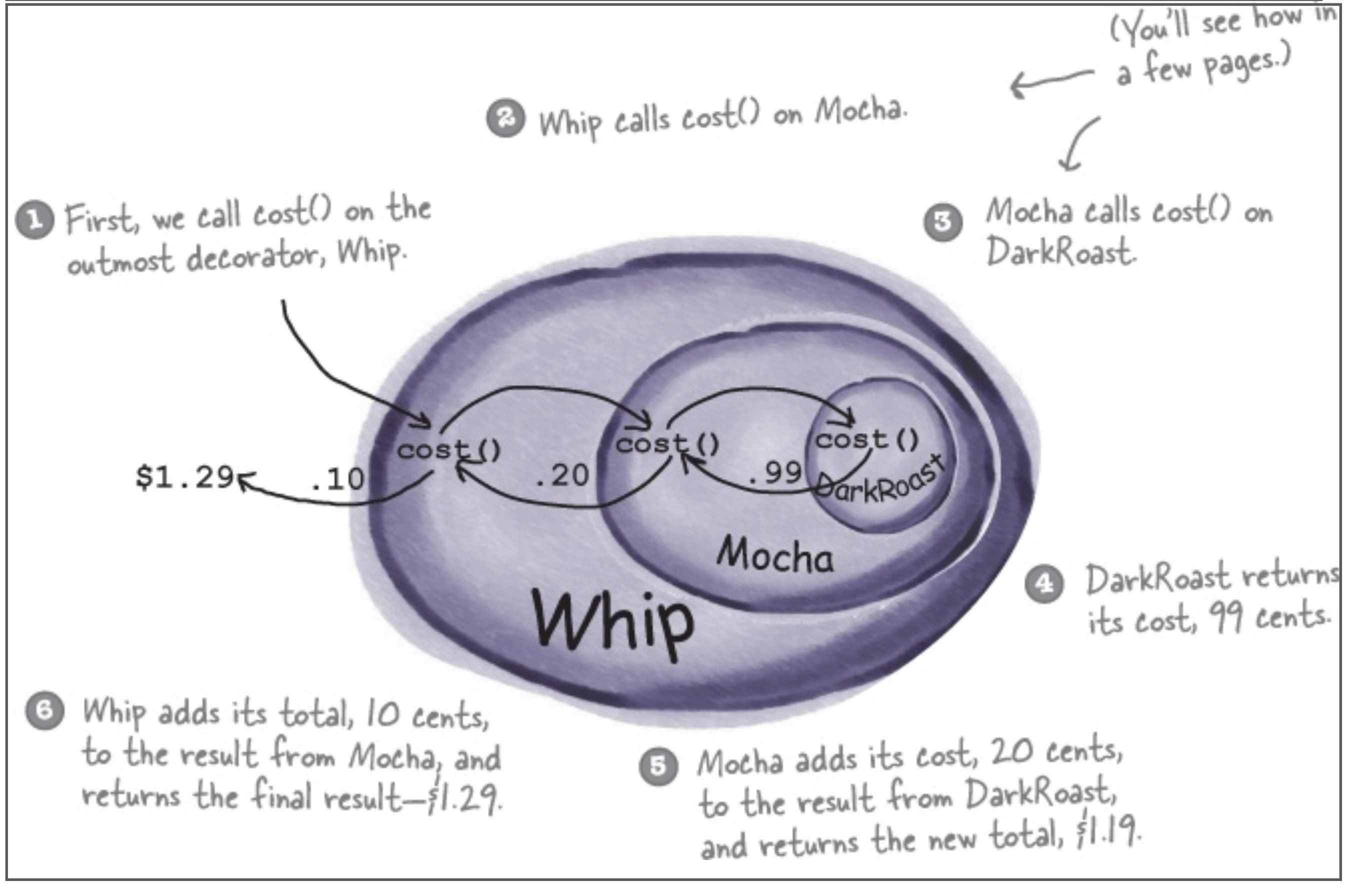
Decorator Pattern: Example

Constructing a drink order with Decorators









Decorator Pattern: Code

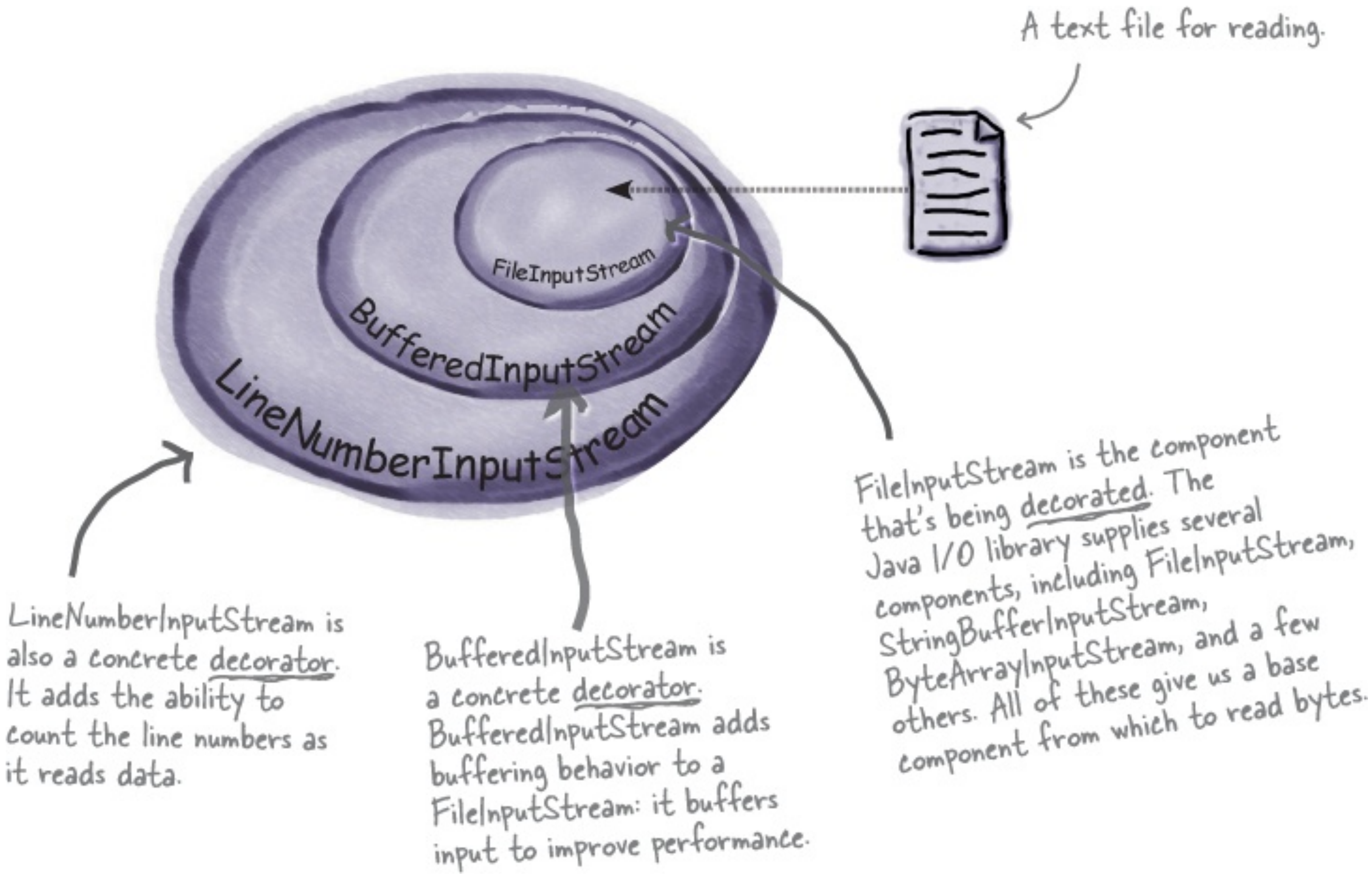
```
Beverage beverage = new Espresso();
System.out.println(beverage.getDescription()
        + " $" + beverage.cost());
System.out.println("-------------------------
Beverage beverage2 = new DarkRoast();
beverage2 = new Mocha(beverage2);
beverage2 = new Mocha(beverage2);
beverage2 = new Whip(beverage2);
System.out.println(beverage2.getDescription()
        + " $" + beverage2.cost());

System.out.println("------------------------- ");

Beverage beverage3 = new HouseBlend();
beverage3 = new Soy(beverage3);
beverage3 = new Mocha(beverage3);
beverage3 = new Whip(beverage3);
System.out.println(beverage3.getDescription()
        + " $" + beverage3.cost());
System.out.println("-------------------------
```

```
public double cost() {
    double beverage_cost = beverage.cost();
    System.out.println("Whipe: beverage.cost() is: " + beverage_cost);
    System.out.println("  - adding One Whip cost of 0.10c ");
    System.out.println("  - new cost is: " + (0.10 + beverage_cost ) );

    return 0.10 + beverage_cost ;
}
```

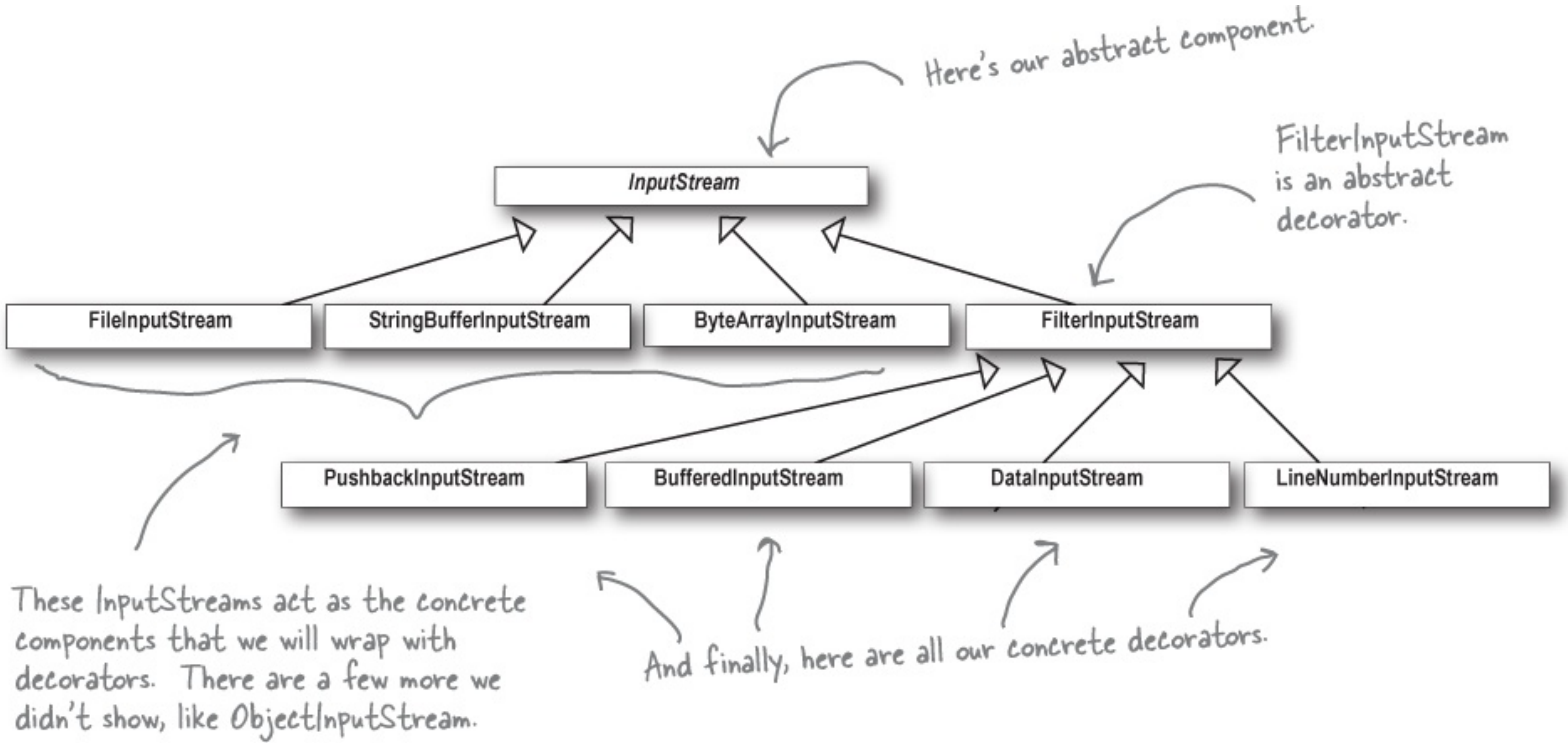
Read the example code discussed/developed in the lectures, and also provided for this week

```
public double cost() {
    double beverage_cost = beverage.cost();
    System.out.println("Mocha: beverage.cost() is: " + beverage_cost );
    System.out.println("  - adding One Mocha cost of 0.20c ");
    System.out.println("  - new cost is: " + (0.20 + beverage_cost ) );

    return 0.20 + beverage_cost ;
}
```

Decorator Pattern: Java I/O Example



Decorator Pattern: Java I/O Example



Decorator Pattern: Code

```
InputStream f1 = new FileInputStream(filename);
InputStream b1 = new BufferedInputStream(f1);
InputStream lCase1 = new LowerCaseInputStream(b1);
InputStream rot13 = new Rot13(b1);

while ((c = rot13.read()) >= 0) {
    System.out.print((char) c);
}
```

Read the example code
discussed/developed in the lectures,
and also provided for this week

Decorator Pattern:

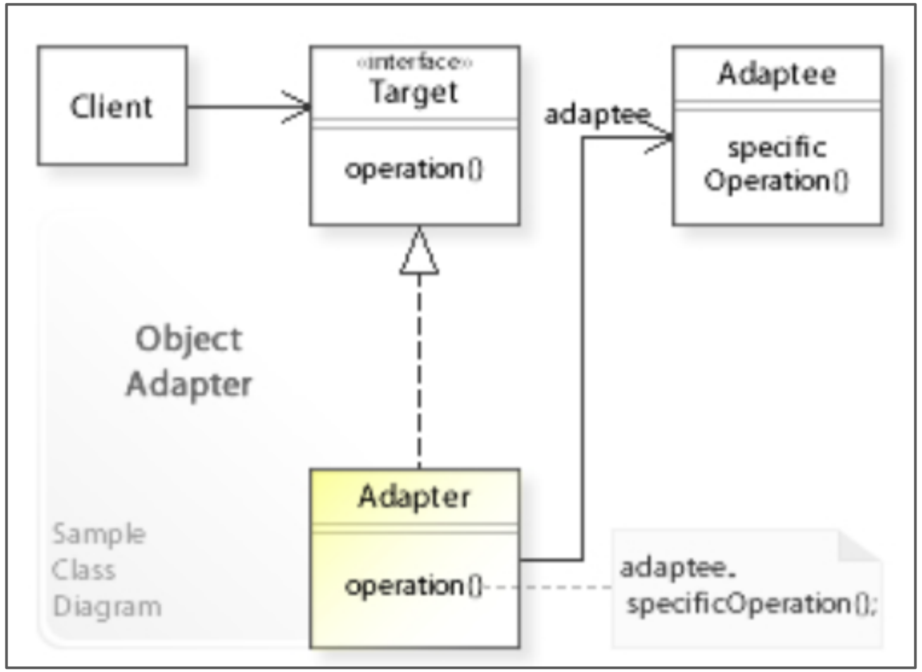
- Demo …

Adapter Pattern

Adapter Pattern : Intent

- "*Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.*" [GoF]
- The adapter pattern allows the interface of an existing class to be used as another interface, suitable for a client class.
- The adapter pattern is often used to make existing classes (APIs) work with a client class without modifying their source code.
- The adapter class maps / joins functionality of two different types / interfaces.
- The adapter patter offers a wrapper around an existing useful class, such that a client class can use functionality of the existing class.
- The adapter pattern do not offer additional functionality.

Adapter Pattern: Structure



- The adapter contains an instance of the class it wraps.
- In this situation, the adapter makes calls to the instance of the wrapped object.

Adapter: Example

```java
interface LightningPhone {
    void recharge();
    void useLightning();
}

interface MicroUsbPhone {
    void recharge();
    void useMicroUsb();
}
```

```java
class Iphone implements LightningPhone {
    private boolean connector;

    @Override
    public void useLightning() {
        connector = true;
        System.out.println("Lightning connected");
    }

    @Override
    public void recharge() {
        if (connector) {
            System.out.println("Recharge started");
            System.out.println("Recharge finished");
        } else {
            System.out.println("Connect Lightning first");
        }
    }
}
```

```java
class Android implements MicroUsbPhone {
    private boolean connector;

    @Override
    public void useMicroUsb() {
        connector = true;
        System.out.println("MicroUsb connected");
    }

    @Override
    public void recharge() {
        if (connector) {
            System.out.println("Recharge started");
            System.out.println("Recharge finished");
        } else {
            System.out.println("Connect MicroUsb first");
        }
    }
}
```

Adapter: Example

```java
class LightningToMicroUsbAdapter implements MicroUsbPhone {
    private final LightningPhone lightningPhone;

    public LightningToMicroUsbAdapter(LightningPhone lightningPhone) {
        this.lightningPhone = lightningPhone;
    }

    @Override
    public void useMicroUsb() {
        System.out.println("MicroUsb connected");
        lightningPhone.useLightning();
    }

    @Override
    public void recharge() {
        lightningPhone.recharge();
    }
}
```

```java
public class AdapterDemo {
    static void rechargeMicroUsbPhone(MicroUsbPhone phone) {
        phone.useMicroUsb();
        phone.recharge();
    }

    static void rechargeLightningPhone(LightningPhone phone) {
        phone.useLightning();
        phone.recharge();
    }

    public static void main(String[] args) {
        Android android = new Android();
        Iphone iPhone = new Iphone();

        System.out.println("Recharging android with MicroUsb");
        rechargeMicroUsbPhone(android);

        System.out.println("Recharging iPhone with Lightning");
        rechargeLightningPhone(iPhone);

        System.out.println("Recharging iPhone with MicroUsb");
        rechargeMicroUsbPhone(new LightningToMicroUsbAdapter(iPhone));
    }
}
```

Output

```
Recharging android with MicroUsb
MicroUsb connected
Recharge started
Recharge finished
Recharging iPhone with Lightning
Lightning connected
Recharge started
Recharge finished
Recharging iPhone with MicroUsb
MicroUsb connected
Lightning connected
Recharge started
Recharge finished
```

Design Patterns: Discuss Differences

- Creational Patterns
  - Abstract Factory
  - Factory Method
  - Singleton
- Structural Patterns
  - Adapter *discussed*
  - Composite *discussed*
  - Decorator *discussed*
- Behavioral Patterns
  - Iterator *discussed*
  - Observer *discussed*
  - State *discussed*
  - Strategy *discussed*
  - Template
  - Visitor

We plan to discuss the rest of the design patterns above in the following weeks; and many more other topics.

End