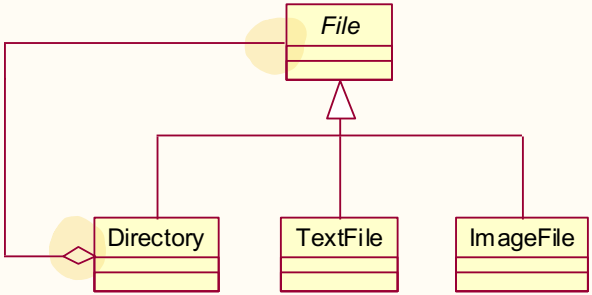
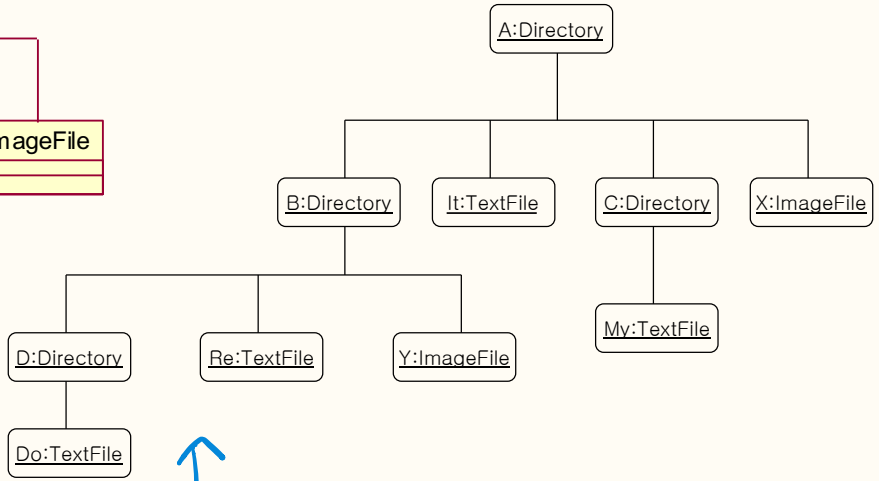
Decorator Pattern

Before we start…

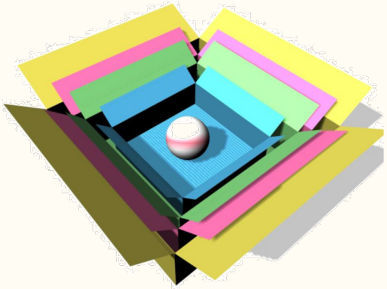


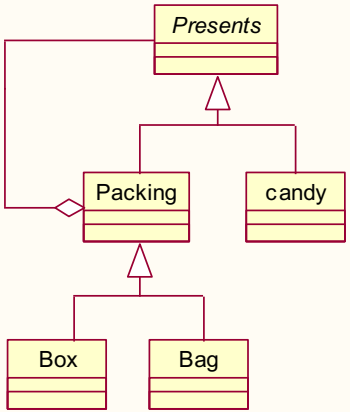
디렉토리는 여러개의 File을 갖는다.



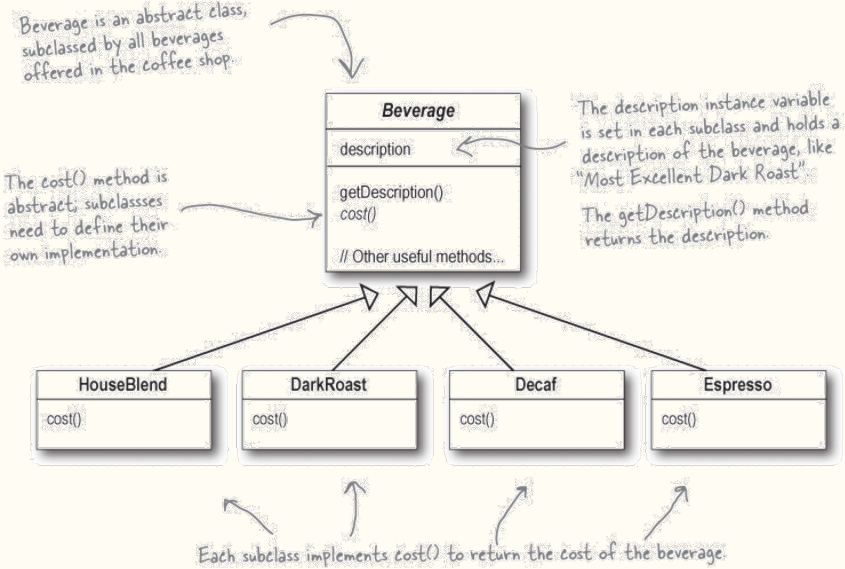
More Practice…

Box안에 Box
계속 Box로 한줄로 내려가게
만들 수 있다.





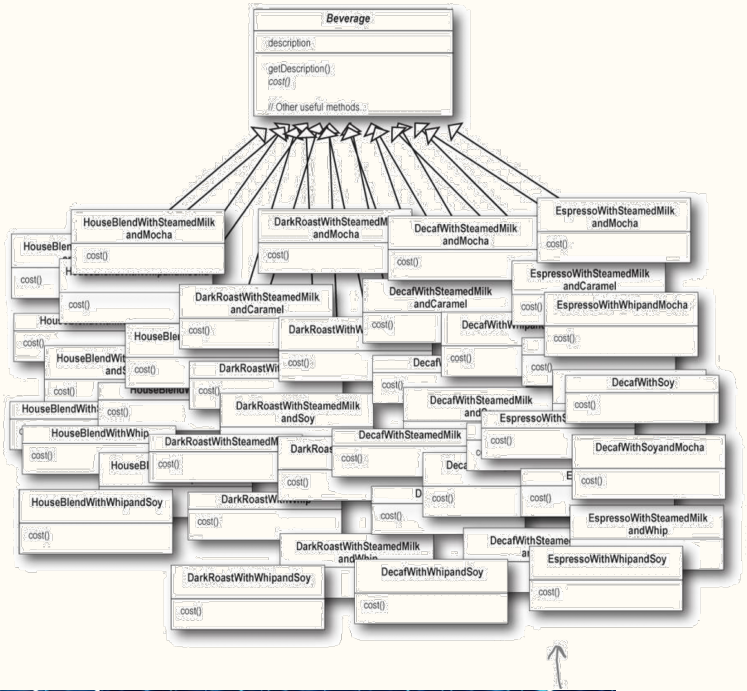
Coffee Ordering System



- Fastest growing coffee shop around
- They've grown so quickly they're scrambling to update their ordering systems to match their beverage offerings.
- Different cost() implementation is required

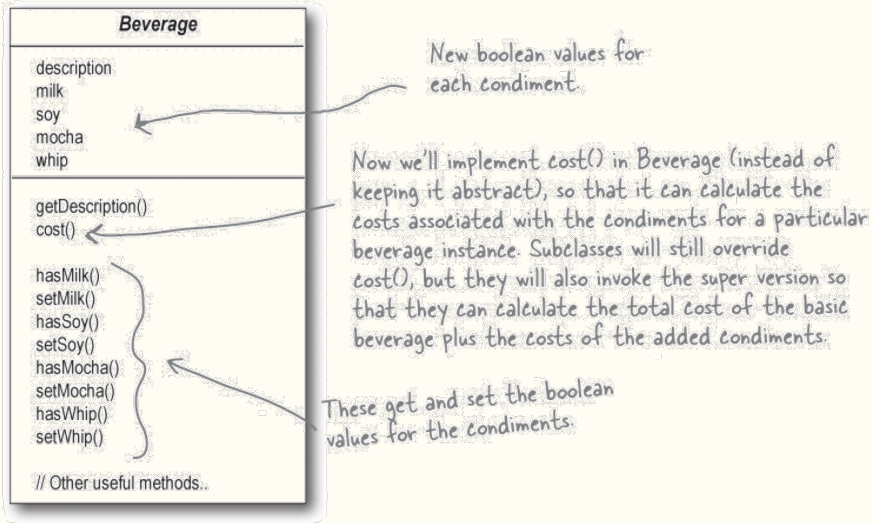
Beverage explosion

- For various condiments such as steamed milk(latte), soy, and mocha, we must make many kinds for all the combination of condiments.
- Each cost() method computes the cost of the coffee along with the other condiments in the order



First attempt

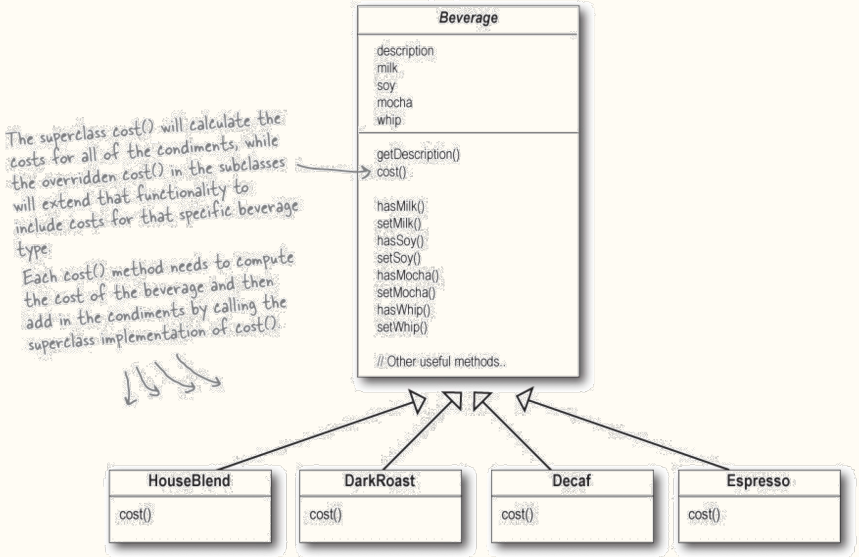
- Let's start with the Beverage base class and add instance variables to represent whether or not each beverage has milk, soy, mocha and whip..
- By setting condiment from subclass, super class can calculate various cost



Changes !!!

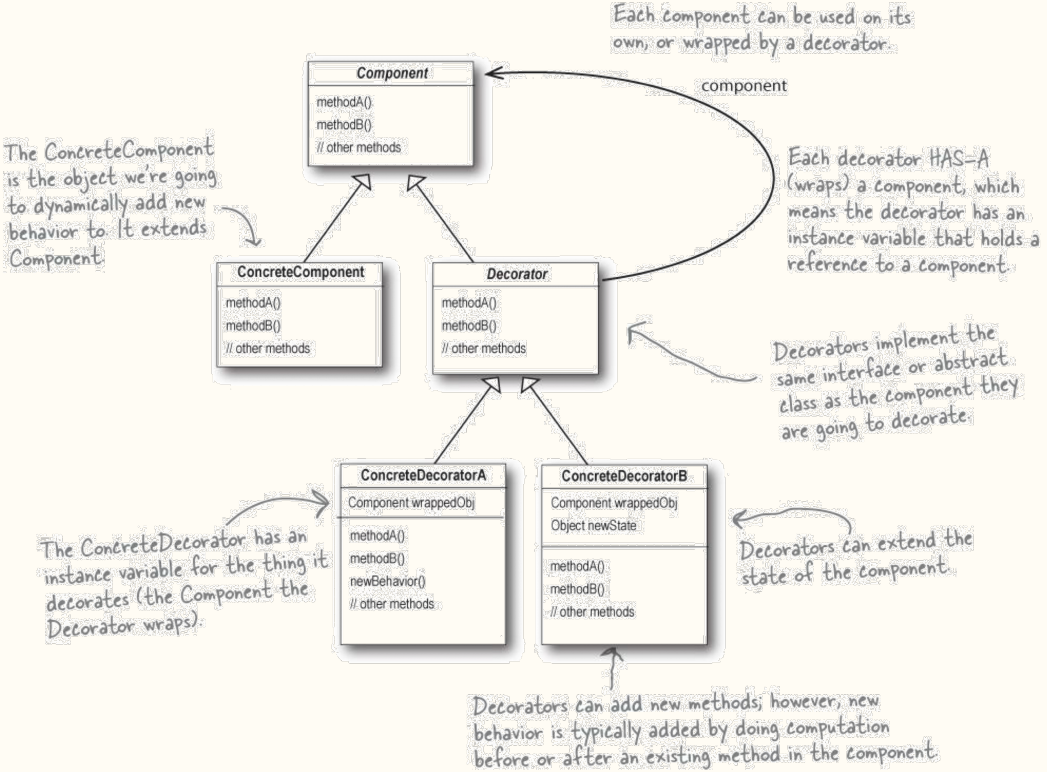
- Price can change, ingredient can be added.
- What happens when the price of milk goes up?
- What do they do when they add a new caramel topping?

- Violating OCP & ISP



Decorator pattern

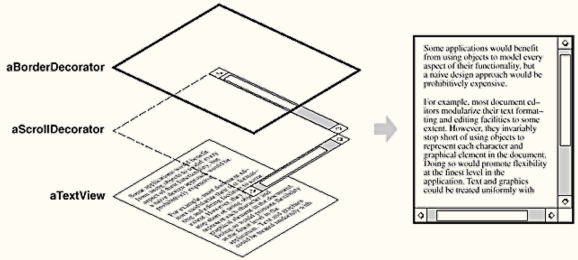
- Intention: Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclass for extending functionality
- Motivation: The object that you want to use does the basic functions you require. However, you may need to add some additional functionality to the object, occurring before or after object's basic functionality
- Solution: Allows for extending the functionality of an object without resorting to sub-classing

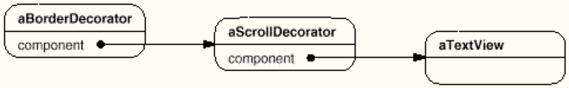

Each component can be used on its own, or wrapped by a decorator.
Component
methodA()
methodB()
// other methods
component
The ConcreteComponent is the object we're going to dynamically add new behavior to. It extends Component.
Each decorator HAS-A (wraps) a component, which means the decorator has an instance variable that holds a reference to a component.
ConcreteComponent
methodA()
methodB()
// other methods
Decorator
methodA()
methodB()
// other methods
Decorators implement the same interface or abstract class as the component they are going to decorate.
ConcreteDecoratorA
Component wrappedObj
methodA()
methodB()
newBehavior()
// other methods
ConcreteDecoratorB
Component wrappedObj
Object newState
methodA()
methodB()
// other methods
The ConcreteDecorator has an instance variable for the thing it decorates (the Component the Decorator wraps).
Decorators can extend the state of the component.
Decorators can add new methods; however, new behavior is typically added by doing computation before or after an existing method in the component.

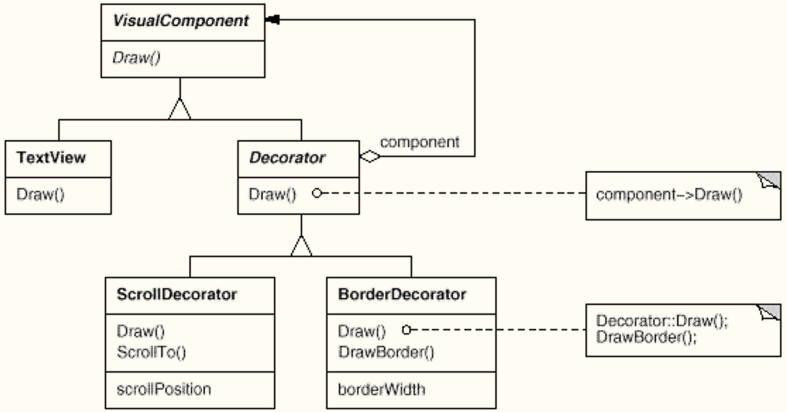
Consequence of decorator pattern

- Functionality that is to be added resides in small objects. The advantage is the ability to dynamically add this function before or after the functionality in the Concrete Component
- Although a decorator may add its functionality before or after that which it decorates, the chain of instantiation always ends with the Concrete Component

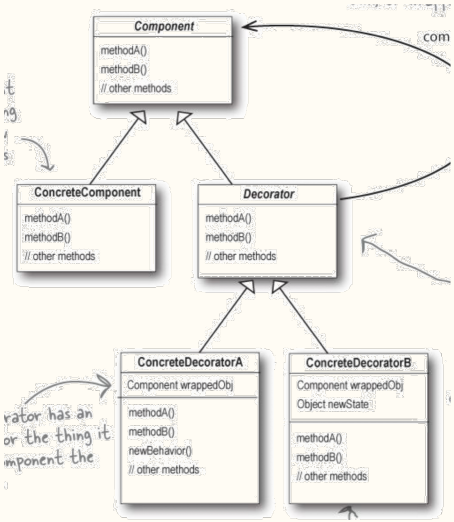
Example

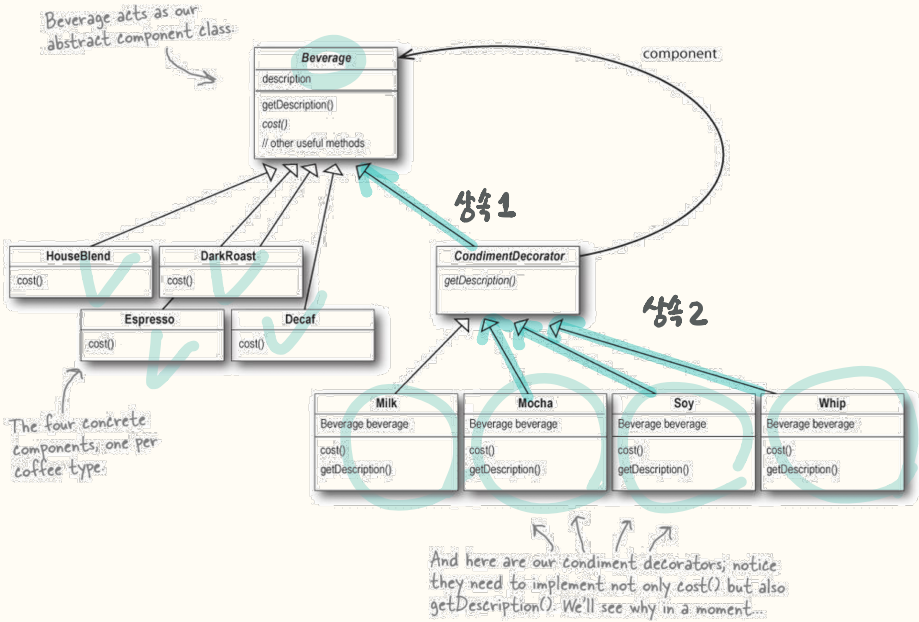

aBorderDecorator
aScrollDecorator
aTextView
Some applications would benefit from using objects to model every aspect of their functionality, but a naive design approach would be prohibitively expensive.
For example, most document editors modularize their text formatting and editing facilities to some extent. However, they invariably stop short of using objects to represent each character and graphical element in the document. Doing so would promote flexibility at the finest level in the application. Text and graphics could be treated uniformly with

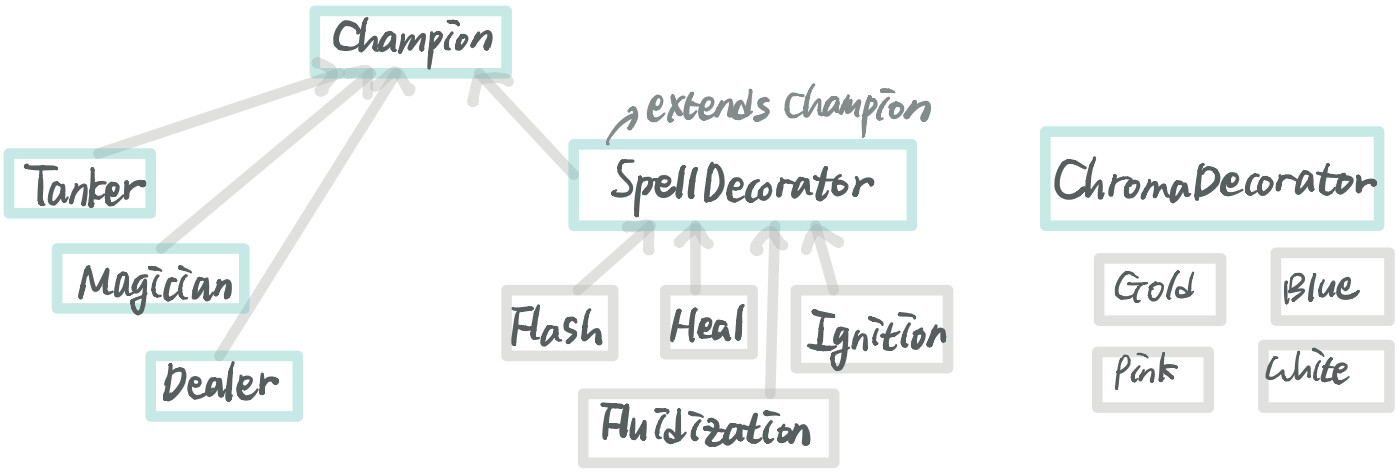

aBorderDecorator
component
aScrollDecorator
component
aTextView


VisualComponent
Draw()
component
TextView
Draw()
Decorator
Draw()
component->Draw()
ScrollDecorator
Draw()
ScrollTo()
scrollPosition
BorderDecorator
Draw()
DrawBorder()
borderWidth
Decorator::Draw();
DrawBorder();

Applying Decorator Pattern







AbilityDecorator

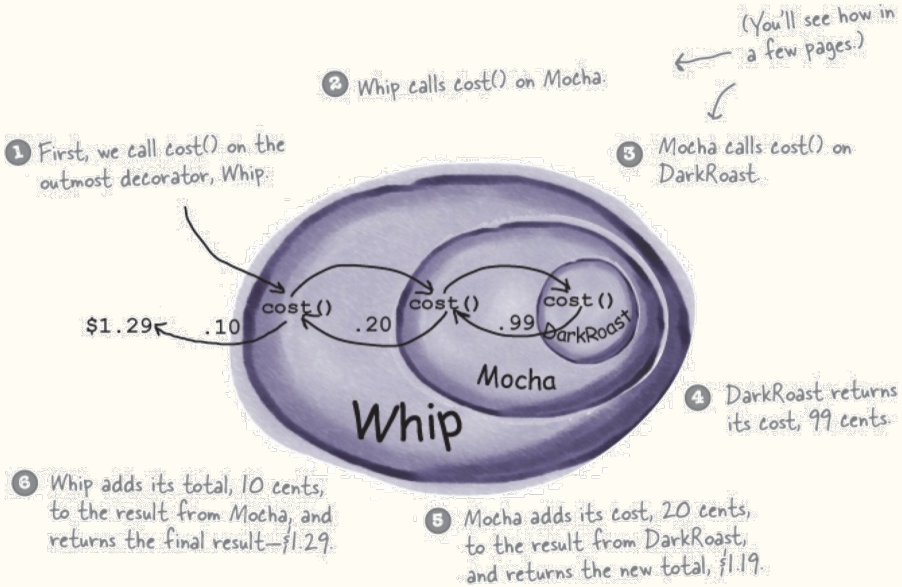
유성 → meteor ability

강전 → electric ability

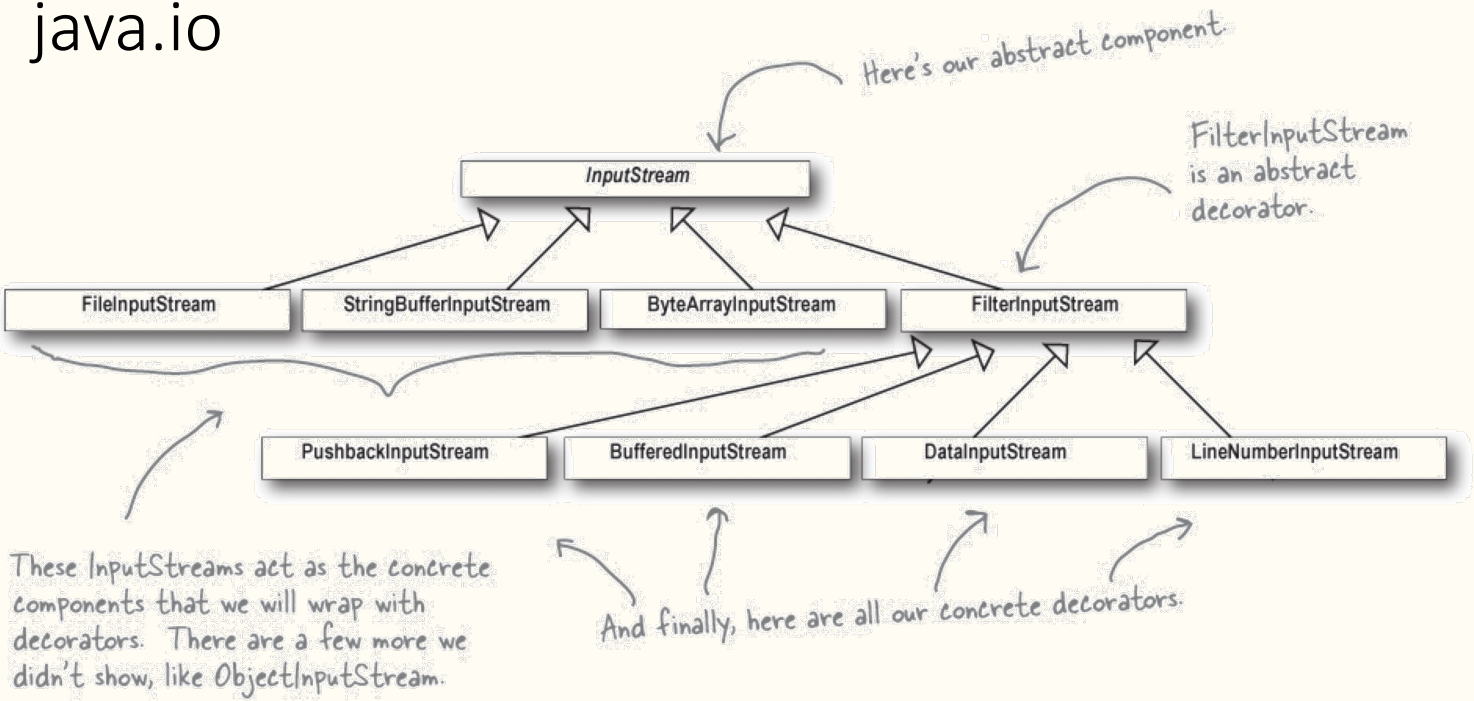
기쁠 → additional speed

벼방패 → additional stamina

As a result, …


(You'll see how in a few pages.)
2 Whip calls cost() on Mocha.
1 First, we call cost() on the outmost decorator, Whip.
3 Mocha calls cost() on DarkRoast.
$1.29
.10
cost()
.20
cost()
.99
cost()
DarkRoast
Mocha
Whip
4 DarkRoast returns its cost, 99 cents.
6 Whip adds its total, 10 cents, to the result from Mocha, and returns the final result—$1.29.
5 Mocha adds its cost, 20 cents, to the result from DarkRoast, and returns the new total, $1.19.

java.io


Here's our abstract component.
InputStream
FilterInputStream is an abstract decorator.
FileInputStream
StringBufferInputStream
ByteArrayInputStream
FilterInputStream
PushbackInputStream
BufferedInputStream
DataInputStream
LineNumberInputStream
These InputStreams act as the concrete components that we will wrap with decorators. There are a few more we didn't show, like ObjectInputStream.
And finally, here are all our concrete decorators.

```
Reader reader = new FileReader(“test.txt”);
Reader reader = new BufferedReader( new FileReader(“test.txt”)  );
Reader reader = new LineNumberReader(
                        new BufferedReader(
                          new FileReader(“test.txt”)  )  );
Reader reader = new LineNumberReader( new FileReader(“test.txt”)  );

java.net.Socket socket = new Socket(hostname, portNumber);
Reader reader = new LineNumberReader(
                        new BufferedReader(
                              new InputStreamReader(
                                    socket.getInputStream()  )  )  );
```

Don't forget to import java.io... (not shown).

First, 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;
    }
}
```

Now we need to implement two read methods. They take a byte (or an array of bytes) and convert each byte (that represents a character) to lowercase if it's an uppercase character.

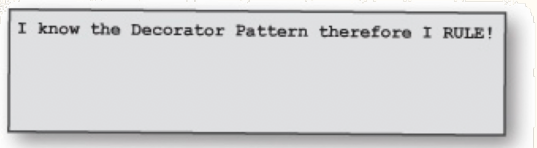
```
public class InputTest {
    public static void main(String[] args) throws IOException {
        int c;

        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();
        }
    }
}
```

Set up the FileInputStream and decorate it, first with a BufferedInputStream and then our brand new LowerCaseInputStream filter.



test.txt file

You need to make this file.

Just use the stream to read characters until the end of file and print as we go.