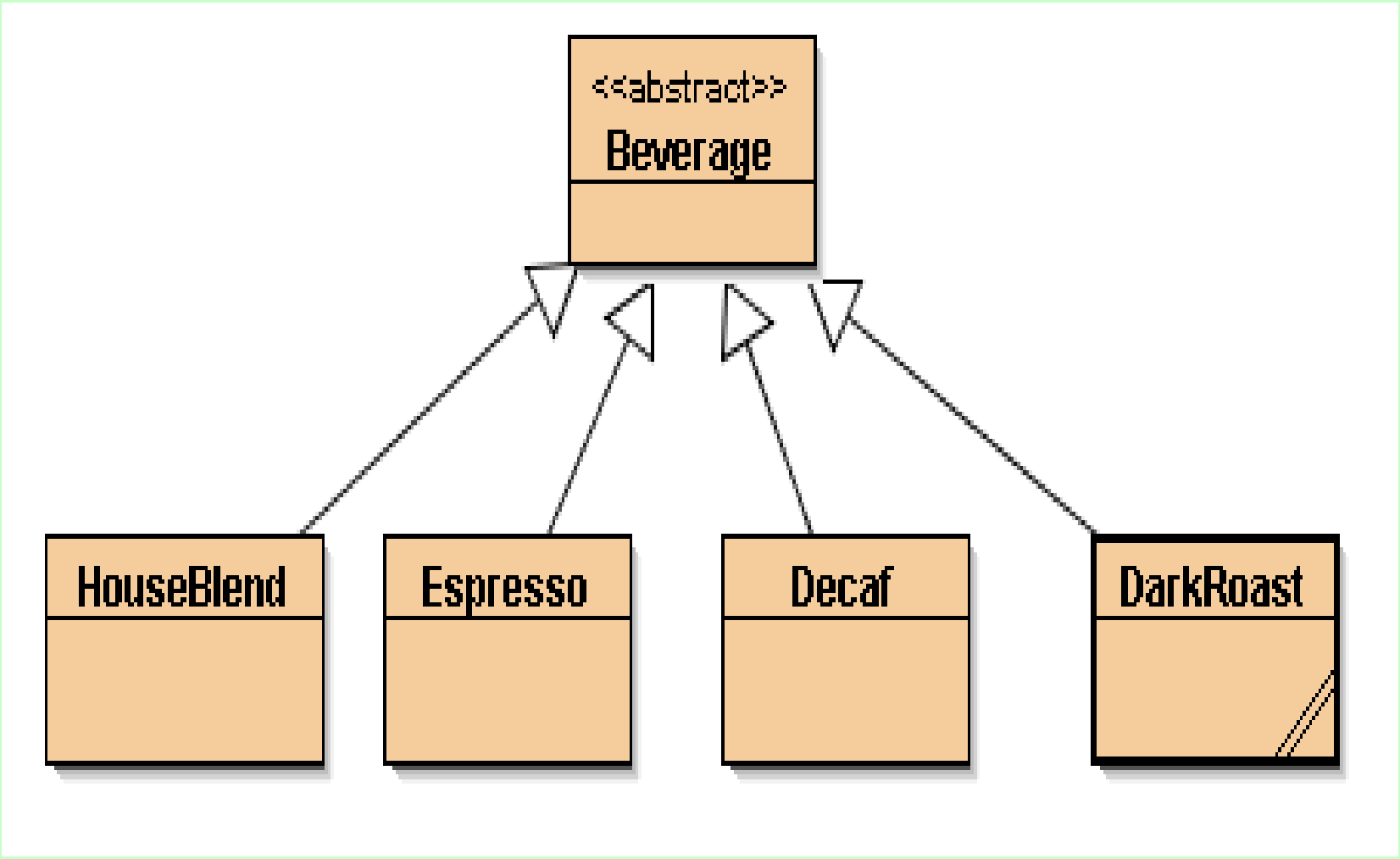
The Decorator Pattern

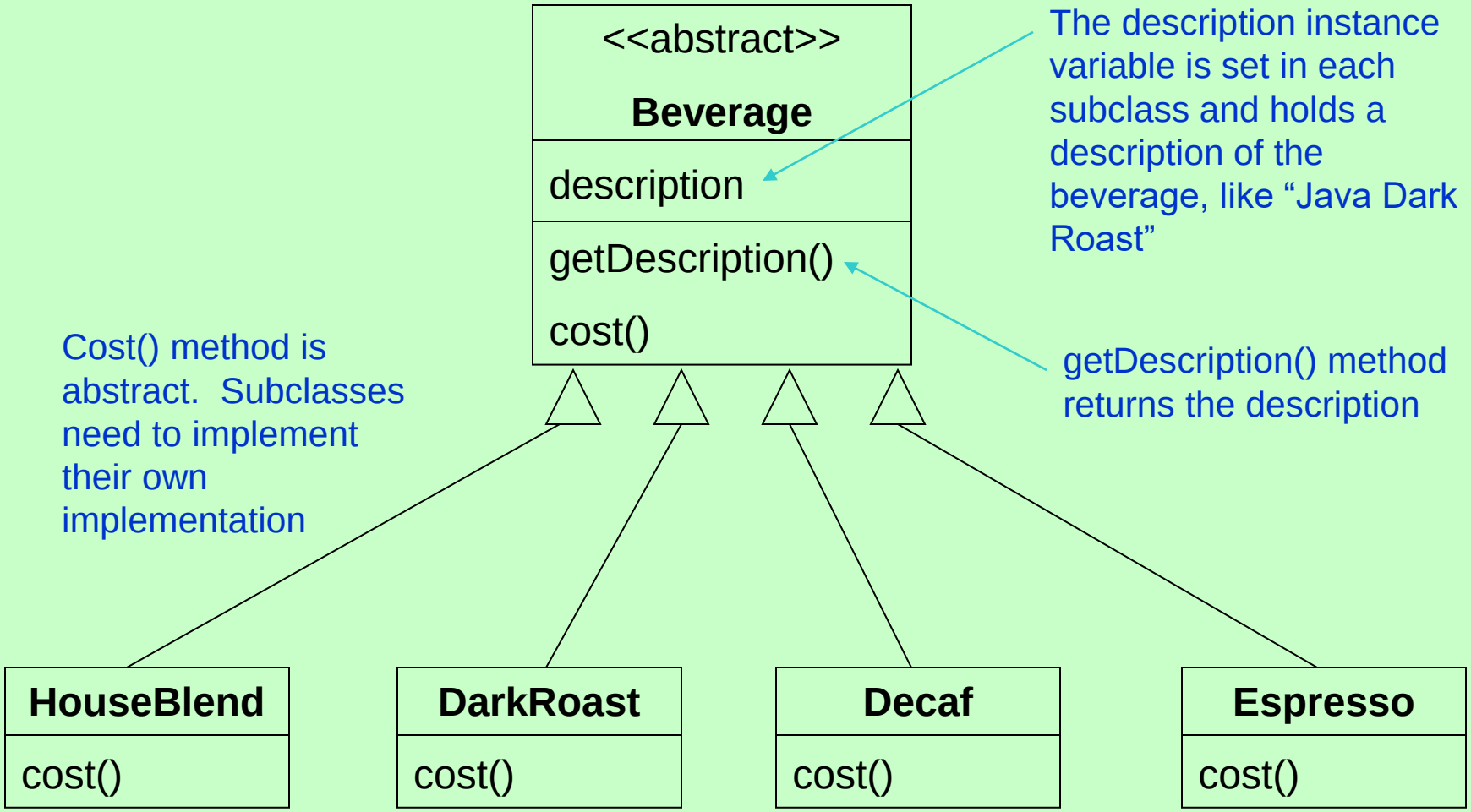
Agenda

- Re-examine the typical overuse of inheritance
- Learn how to decorate classes at runtime using a form of composition
- Will be able to give your objects new responsibilities without making any code change

Problem

- Example: StarBuzz Coffee
  - Several blends
    - HouseBlend, DarkRoast, Decaf, Espresso
  - Condiments
    - Steamed milk, soy, mocha, whipped milk
  - Extra charge for each
  - How do we charge all combinations?
    - First attempt: inheritance


<<abstract>>
Beverage
HouseBlend
Espresso
Decaf
DarkRoast

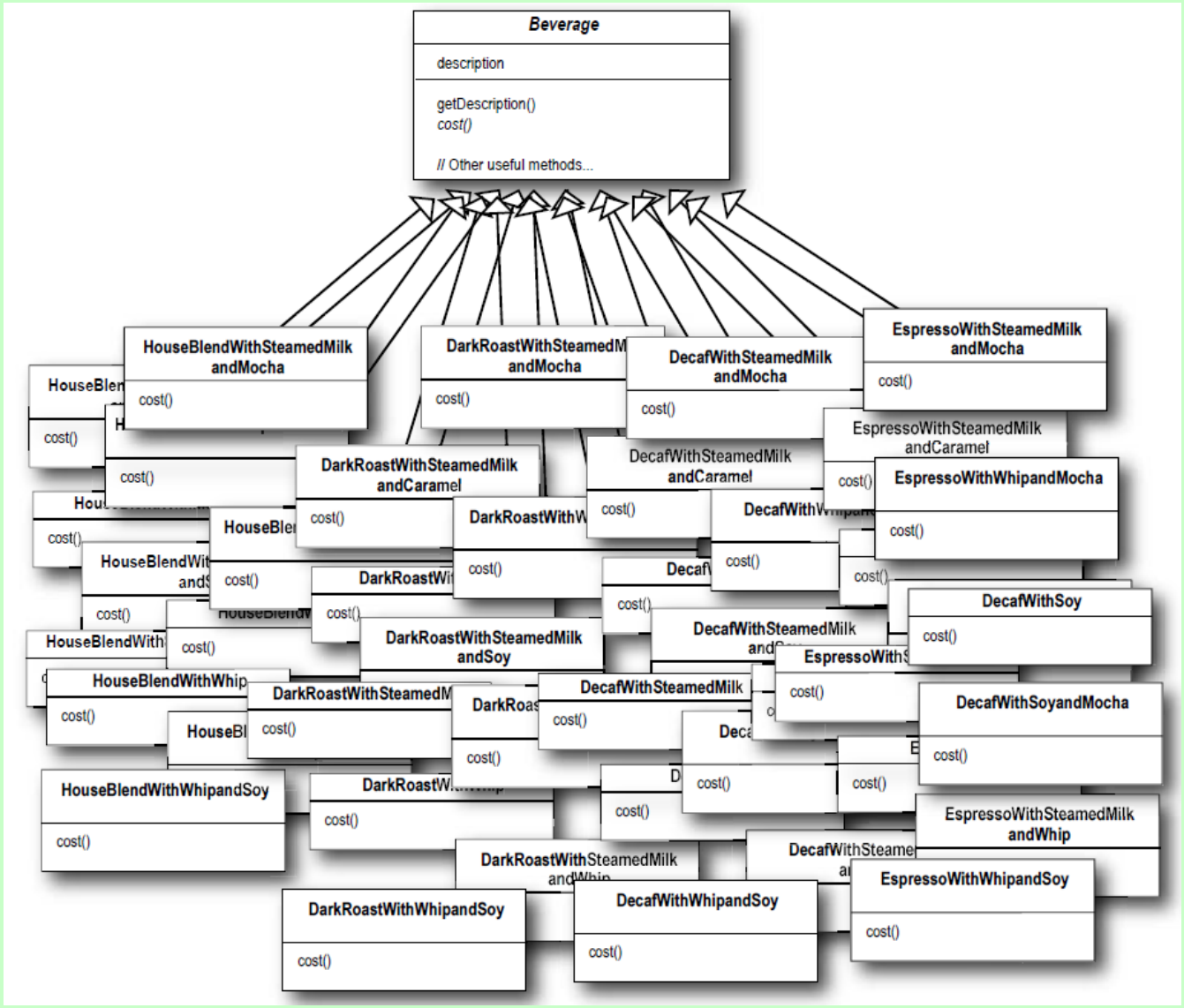


Each subclass implements cost() to return the cost of the beverage

Problem with first attempt:

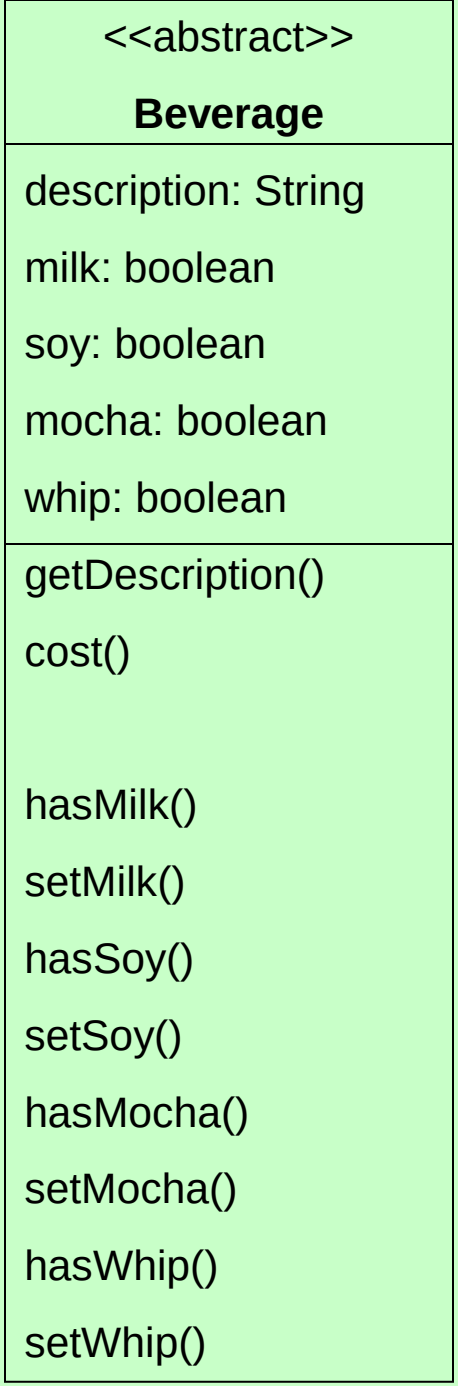
- How do we add the condiments?
  - Apply subclass and inheritance
    - Add subclasses and inheritance for condiments
    - But there can be combinations of condiments

Problem with first attempt:



Problem with first attempt:

- Class explosion when all combinations are considered
  - There must be a better way
  - Can we treat condiments as instance variables and apply inheritance?



New boolean values for each condiment

Beverage's cost() calculates the cost of condiments.

The subclasses calculate the cost of the beverage and add the cost of condiments

We implement cost() in Beverage (instead of keeping it abstract), so that it can calculate the costs associated with the condiments for a particular beverage instance. Subclasses will still override cost(), but they also invoke the super version to calculate the total cost() of the basic beverage plus the costs of the added condiments.

Any problem?

- Our goal is to simplify maintenance.
  - Prices can change – code change
  - New condiments – add new methods and alter cost method in superclass
  - New beverages e.g. tea
  - Double mocha?
- Code changes in the superclass when the above happens or in using combinations

Design Principle

- Open-Closed Principle
- Classes should be open for extension, but closed for modification
  - Apply the principle to the areas that are most likely to change
- We want our designs that are resilient to change and flexible enough to take on new functionality to meet changing requirements

How?

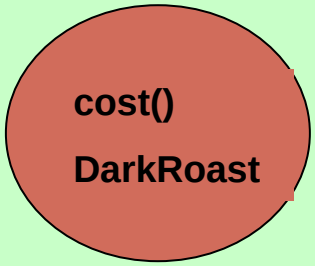
- We want techniques *to allow code to be extended without direct modification*
  - Allow classes to be easily extended to incorporate new behaviour without modifying existing code

Decorator Pattern

- Attaches additional responsibilities to an object dynamically. Decorator provides a flexible alternative to subclassing for extending functionality.
- Example: StarBuzz Coffee
  - Several blends
    - HouseBlend, DarkRoast, Decaf, Espresso
  - Condiments
    - Steamed milk, soy, mocha, whipped milk
  - Extra charge for each
  - How do we charge all combinations?
    - First attempt: inheritance
    - Second attempt: instance variables + inheritance

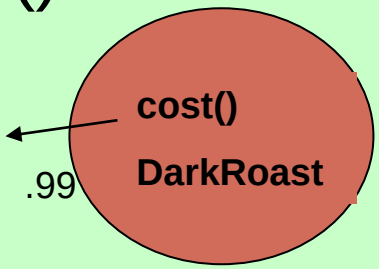
Start with a beverage and decorate it with condiments at run time

- Example:
  - Take a DarkRoast object



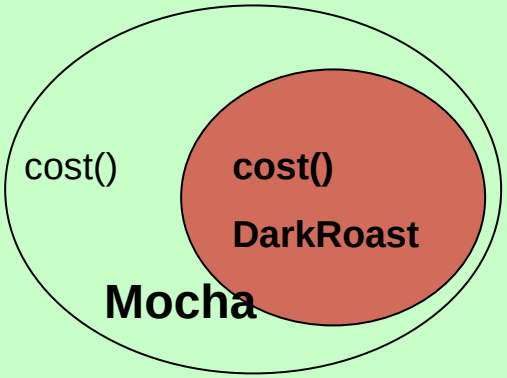
Start with a beverage and decorate it with condiments at run time

- Example:
  - Take a DarkRoast object
  - Call the cost() method



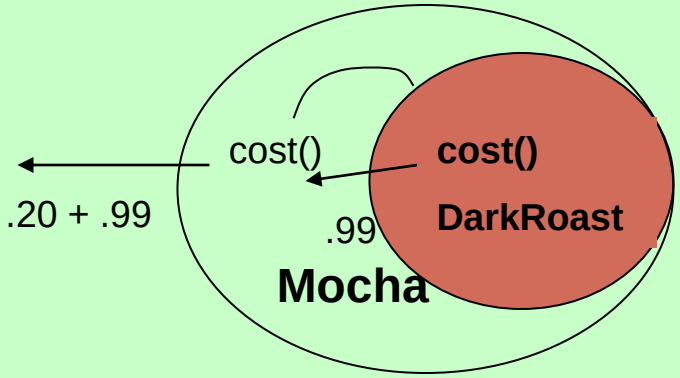
Start with a beverage and decorate it with condiments at run time

- Example:
  - Take a DarkRoast object
  - Decorate it with a Mocha object



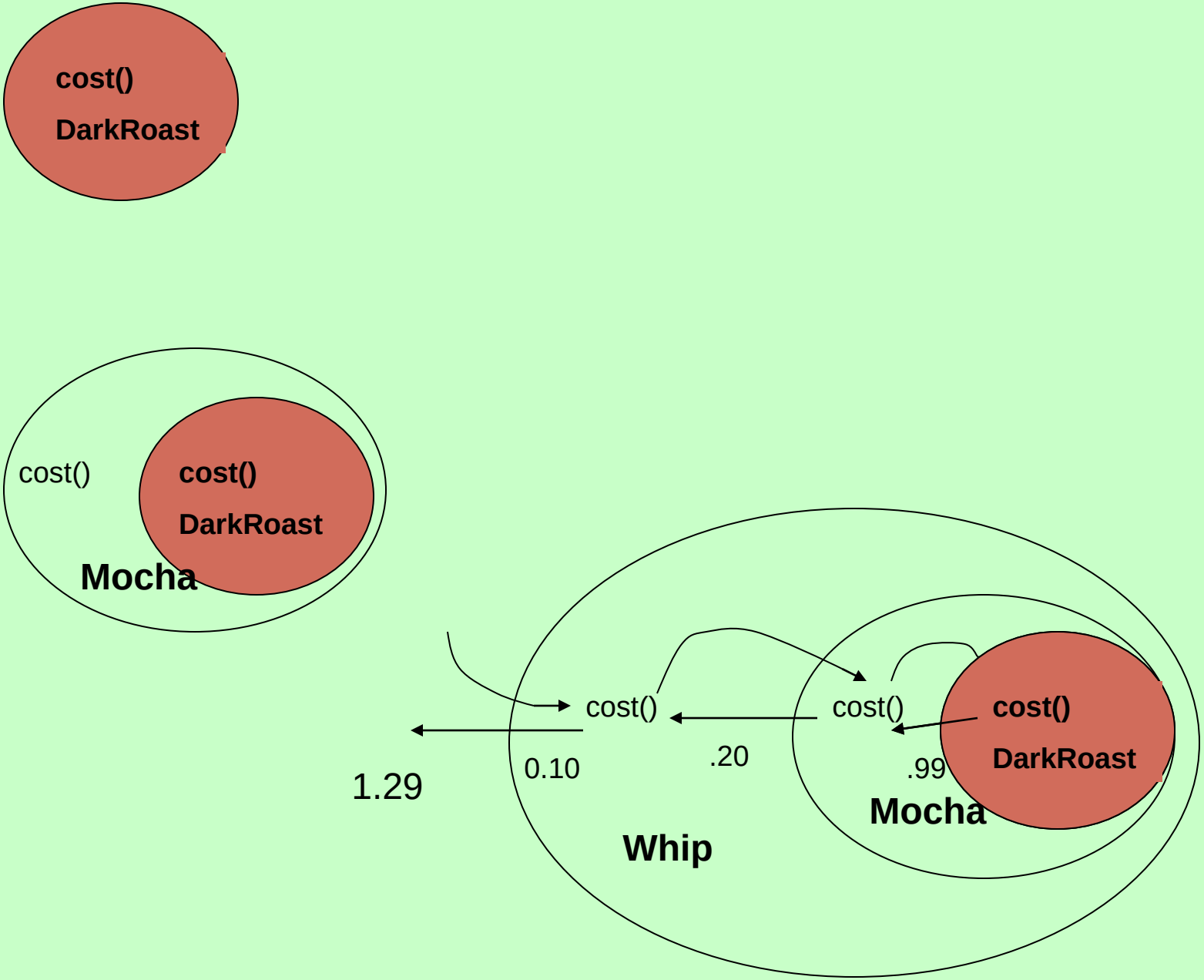
Start with a beverage and decorate it with condiments at run time

- Example:
  - Take a DarkRoast object
  - Decorate it with a Mocha object
  - Call the cost() method and rely on delegation to add the condiment costs



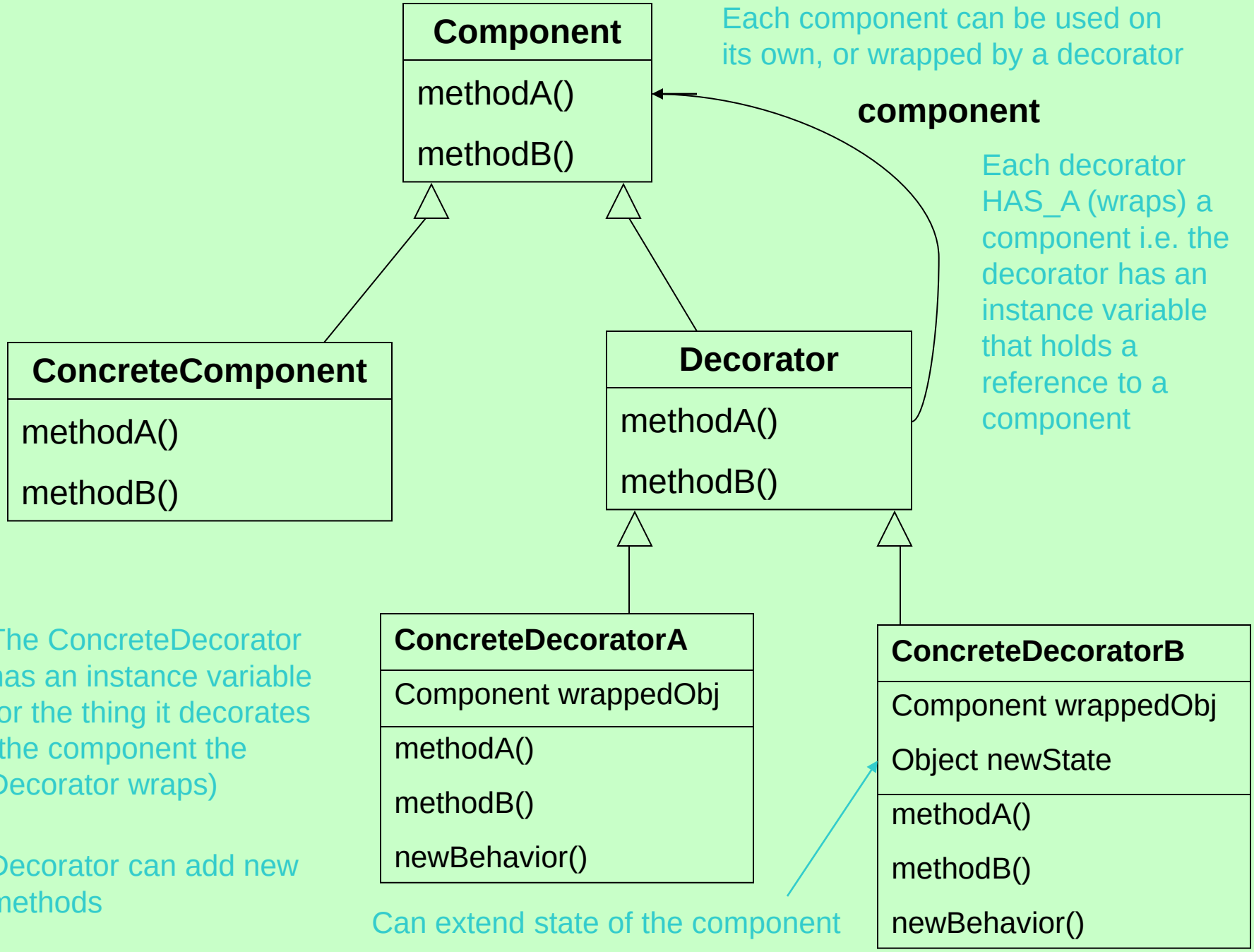
Start with a beverage and decorate it with condiments at run time

- Example:
  - Take a DarkRoast object
  - Decorate it with a Mocha object
  - Decorate it with a Whip object
  - Call the cost() method and rely on delegation to add the condiment costs


cost()
DarkRoast
cost()
cost()
DarkRoast
Mocha
cost()
cost()
cost()
DarkRoast
0.10
.20
.99
1.29
Mocha
Whip

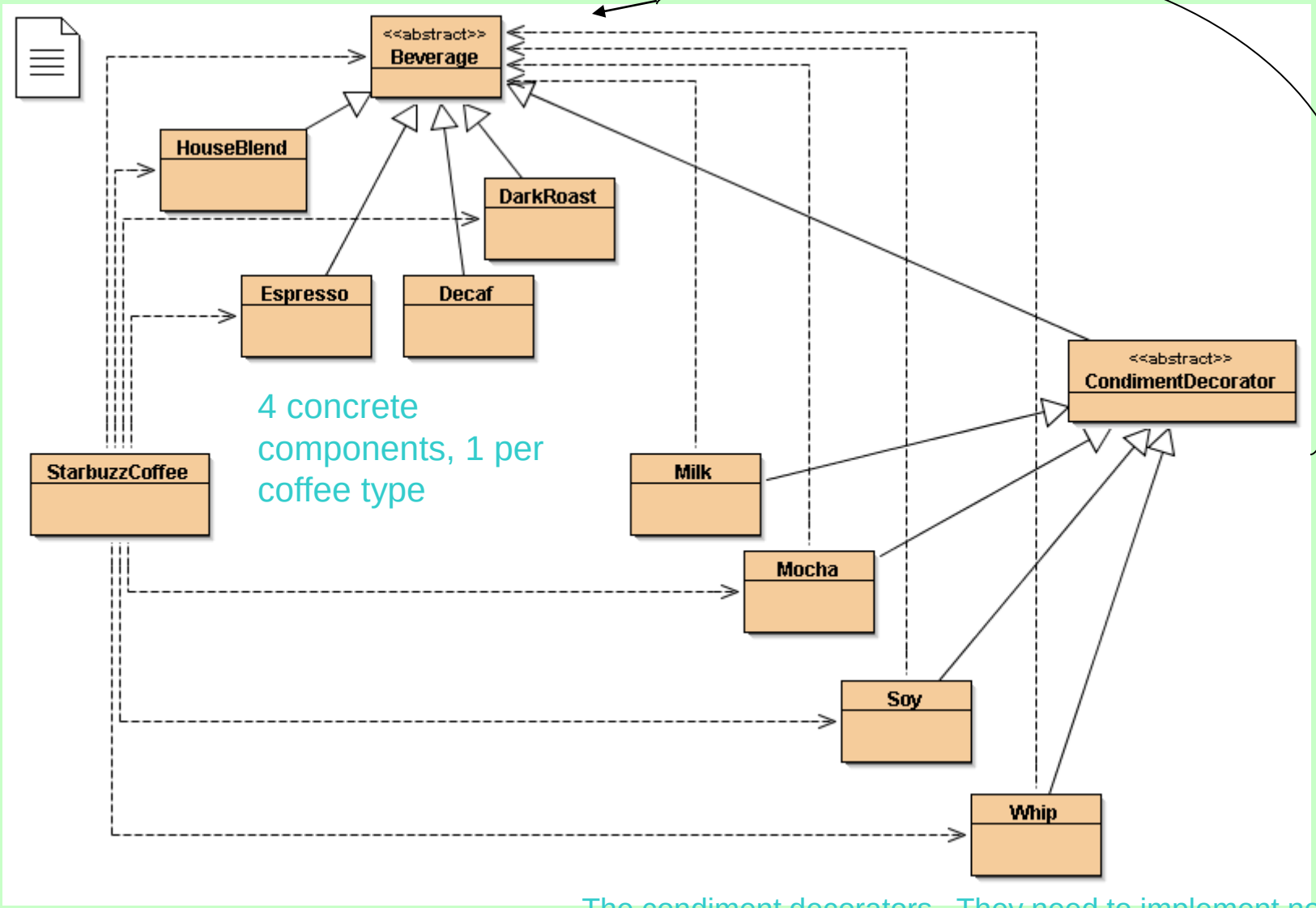
Decorator Pattern

- The Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality
- See class diagram on the next slide


Component
methodA()
methodB()
Each component can be used on its own, or wrapped by a decorator
component
Each decorator HAS_A (wraps) a component i.e. the decorator has an instance variable that holds a reference to a component
ConcreteComponent
methodA()
methodB()
Decorator
methodA()
methodB()
The ConcreteDecorator has an instance variable for the thing it decorates (the component the Decorator wraps)
Decorator can add new methods
ConcreteDecoratorA
Component wrappedObj
methodA()
methodB()
newBehavior()
Can extend state of the component
ConcreteDecoratorB
Component wrappedObj
Object newState
methodA()
methodB()
newBehavior()

Beverage acts as abstract component class

component



4 concrete components, 1 per coffee type

The condiment decorators. They need to implement not only cost() but also getDescription()

Discussion

- It appears that the condimentDecorator is a subclass of Beverage
  - Really we are subclassing to get the correct *type*, not to inherit its behavior
  - When we *compose* a decorator with a component, we are adding new behavior
  - Inheritance: the behavior is *determined statically* at *compile* time
  - Composition: we can *mix and match* decorators any way we like at runtime.

More: Inheritance vs composition

Inheritance

- Need to change existing code any time we want new behavior

Composition

- Can implement new decorators at any time to add new behavior

One more question:

- We started with Beverage as an abstract class in Java. Can it be an interface in Java?
- Discussion: yes or no? Why?

Code Study:Decorator project - Beverage

```
public abstract class Beverage {
        String description = "Unknown Beverage";

        public String getDescription() {     // already implemented
                return description;
        }

        public abstract double cost();       // Need to implement cost()
}
```

Condiments (Decorator) class

We need to be interchangeable with a Beverage, so extend the Beverage class – not to get its behavior but its type

```
public abstract class CondimentDecorator extends Beverage {

        public abstract String getDescription();
}
```

Here we require all the condiment decorators reimplement the getDescription() method. Explanations coming

Coding beverages

```
public class HouseBlend extends Beverage {

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

        public double cost() {
                return .89;
        }
}
```

Note the description variable is inherited from Beverage. To take care of description, put this in the constructor for the class

Compute the cost of a HouseBlend. Need not worry about condiments

Coding condiments

Mocha is decorator, so extend CondimentDecorator, which extends Beverage

```
public class Mocha extends CondimentDecorator {
    Beverage beverage;

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

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

    public double cost() {
        return .20 + beverage.cost();
    }
}
```

Cost of condiment + cost of beverage

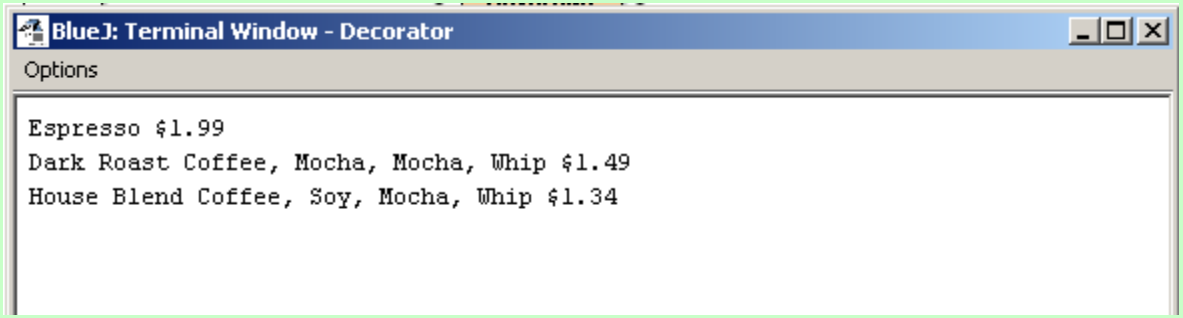
We want the description to include the beverage – say Houseblend – and the condiments

Instantiate Mocha with a reference to a Beverage using

1. An instance variable to hold the beverage we are wrapping
2. A way to set this instance to the object we are wrapping – we pass the beverage we are wrapping to the decorator's constructor

```
public class StarbuzzCoffee {

    public static void main(String args[]) {
        Beverage beverage = new Espresso();  // espresso order, no condiments
        System.out.println(beverage.getDescription()
                + " $" + beverage.cost());

        Beverage beverage2 = new DarkRoast();          // get a DarkRoast
        beverage2 = new Mocha(beverage2);              // wrap it with Mocha
        beverage2 = new Mocha(beverage2);              // warp it with Mocha
        beverage2 = new Whip(beverage2);               // Wrap it with a Whip
        System.out.println(beverage2.getDescription()
                + " $" + beverage2.cost());

        Beverage beverage3 = new HouseBlend();         // get a Houseblend
        beverage3 = new Soy(beverage3);                // wrap with Soy
        beverage3 = new Mocha(beverage3);              // wrap with Mocha
        beverage3 = new Whip(beverage3);               // wrap with Whip
        System.out.println(beverage3.getDescription()
                + " $" + beverage3.cost());
    }
}
```

Executing StarBuzzCoffee



Summary

1. CondimentDecorator extends Beverage class
   - Purpose: the decorators have the same type as the objects they are decorating
   - Inheritance is used to achieve *type matching*
   - Inheritance is NOT used to *get behavior*
2. Where does the *behavior* come in*?*
   - We acquire *behavior* not by inheriting from a superclass, but by *composing objects* together

Summary

3. We subclass the abstract class Beverage to have the correct type not to inherit its behavior
4. The behavior comes in through the composition of decorators with the base components as well as other decorators
5. Because we are using object composition, we get a more flexibility about how to mix and match condiments and beverages.
6. Inheritance determines the behavior at *compile time*. With composition, we can mix and match decorators any way at will at *runtime.*

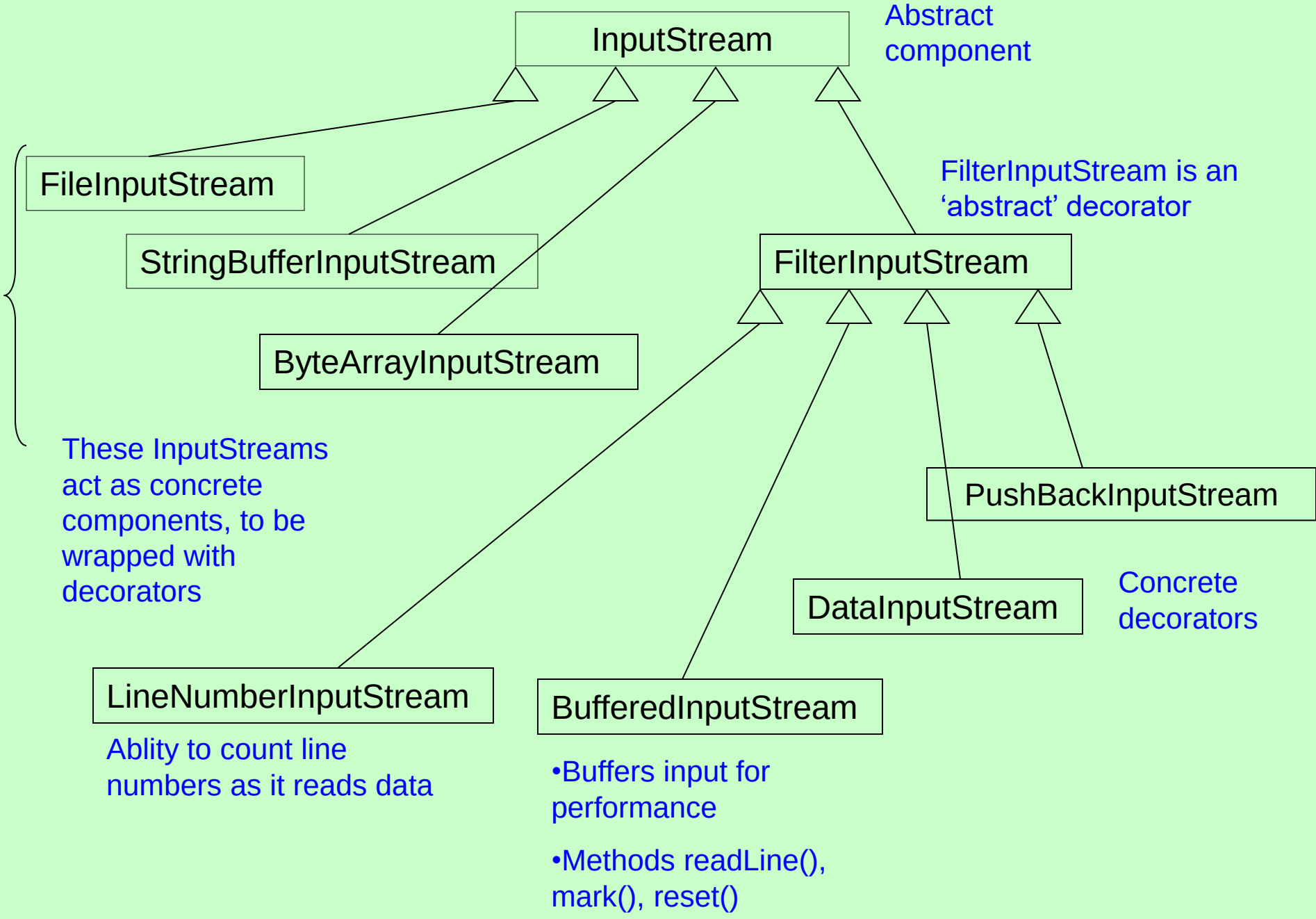
Summary

7. We started with Beverage as an abstract class in Java. Could it be an interface in Java?

Since all we need to inherit is the type of component, we could use an interface in Java.

Decorator

Decorating the java.io classes


InputStream
Abstract component
FileInputStream
StringBufferInputStream
ByteArrayInputStream
These InputStreams act as concrete components, to be wrapped with decorators
FilterInputStream is an 'abstract' decorator
FilterInputStream
PushBackInputStream
DataInputStream
Concrete decorators
LineNumberInputStream
Ablity to count line numbers as it reads data
BufferedInputStream
•Buffers input for performance
•Methods readLine(), mark(), reset()

java.io

Class InputStream

java.lang.Object
  |
  +--**java.io.InputStream**

Direct Known Subclasses:

AudioInputStream, ByteArrayInputStream, FileInputStream, FilterInputStream, InputStream, ObjectInputStream, PipedInputStream, SequenceInputStream, StringBufferInputStream

---

public abstract class **InputStream**
extends Object

This abstract class is the superclass of all classes representing an input stream of bytes.

Applications that need to define a subclass of `InputStream` must always provide a method that returns the next byte of input.

Since:

JDK1.0

See Also:

`BufferedInputStream`, `ByteArrayInputStream`, `DataInputStream`, `FilterInputStream`, `read()`, `OutputStream`, `PushbackInputStream`

java.io

Class FilterInputStream

```
java.lang.Object
  |
  +--java.io.InputStream
        |
        +--java.io.FilterInputStream
```

Direct Known Subclasses:

BufferedInputStream, CheckedInputStream, DataInputStream, DigestInputStream, InflaterInputStream, LineNumberInputStream, ProgressMonitorInputStream, PushbackInputStream

---

public class **FilterInputStream**
extends InputStream

A `FilterInputStream` contains some other input stream, which it uses as its basic source of data, possibly transforming the data along the way or providing additional functionality. The class `FilterInputStream` itself simply overrides all methods of `InputStream` with versions that pass all requests to the contained input stream. Subclasses of `FilterInputStream` may further override some of these methods and may also provide additional methods and fields.

Since:

JDK1.0

java.io

Class BufferedInputStream

```
java.lang.Object
   |
   +--java.io.InputStream
         |
         +--java.io.FilterInputStream
               |
               +--java.io.BufferedInputStream
```

---

public class **BufferedInputStream**
extends FilterInputStream

A `BufferedInputStream` adds functionality to another input stream-namely, the ability to buffer the input and to support the `mark` and `reset` methods. When the `BufferedInputStream` is created, an internal buffer array is created. As bytes from the stream are read or skipped, the internal buffer is refilled as necessary from the contained input stream, many bytes at a time. The `mark` operation remembers a point in the input stream and the `reset` operation causes all the bytes read since the most recent `mark` operation to be reread before new bytes are taken from the contained input stream.

Since:

JDK1.0

java.io

Class DataInputStream

```
java.lang.Object
  |
  +--java.io.InputStream
        |
        +--java.io.FilterInputStream
              |
              +--java.io.DataInputStream
```

All Implemented Interfaces:

DataInput

---

public class **DataInputStream**
extends FilterInputStream
implements DataInput

A data input stream lets an application read primitive Java data types from an underlying input stream in a machine-independent way. An application uses a data output stream to write data that can later be read by a data input stream.

Data input streams and data output streams represent Unicode strings in a format that is a slight modification of UTF-8. (For more information, see X/Open Company Ltd., "File System Safe UCS Transformation Format (FSS_UTF)", X/Open Preliminary Specification, Document Number: P316. This information also appears in ISO/IEC 10646, Annex P.)

All characters in the range `'\u0001'` to `'\u007F'` are represented by a single byte:

0	bits 0-7

java.io

Class PushbackInputStream

```
java.lang.Object
  |
  +--java.io.InputStream
        |
        +--java.io.FilterInputStream
              |
              +--java.io.PushbackInputStream
```

---

public class **PushbackInputStream**
extends FilterInputStream

A `PushbackInputStream` adds functionality to another input stream, namely the ability to "push back" or "unread" one byte. This is useful in situations where it is convenient for a fragment of code to read an indefinite number of data bytes that are delimited by a particular byte value; after reading the terminating byte, the code fragment can "unread" it, so that the next read operation on the input stream will reread the byte that was pushed back. For example, bytes representing the characters constituting an identifier might be terminated by a byte representing an operator character; a method whose job is to read just an identifier can read until it sees the operator and then push the operator back to be re-read.

Since:

JDK1.0

java.io

Class LineNumberInputStream

```
java.lang.Object
  |
  +--java.io.InputStream
        |
        +--java.io.FilterInputStream
              |
              +--java.io.LineNumberInputStream
```

Deprecated. *This class incorrectly assumes that bytes adequately represent characters. As of JDK 1.1, the preferred way to operate on character streams is via the new character-stream classes, which include a class for counting line numbers.*

public class **LineNumberInputStream**
extends FilterInputStream

This class is an input stream filter that provides the added functionality of keeping track of the current line number.

A line is a sequence of bytes ending with a carriage return character (`'\r'`), a newline character (`'\n'`), or a carriage return character followed immediately by a linefeed character. In all three cases, the line terminating character(s) are returned as a single newline character.

The line number begins at `0`, and is incremented by `1` when a `read` returns a newline character.

Since:
JDK1.0

Writing your own Java I/O decorator

- We have learned the decorator pattern
- And I/O class diagram
- Write a decorator that converts all uppercase characters to lowercase characters in the input stream

Extend the FilterInputStream, the abstract decorator for all inputStream

```
import java.io.*;

public class LowerCaseInputStream extends FilterInputStream {

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

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

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

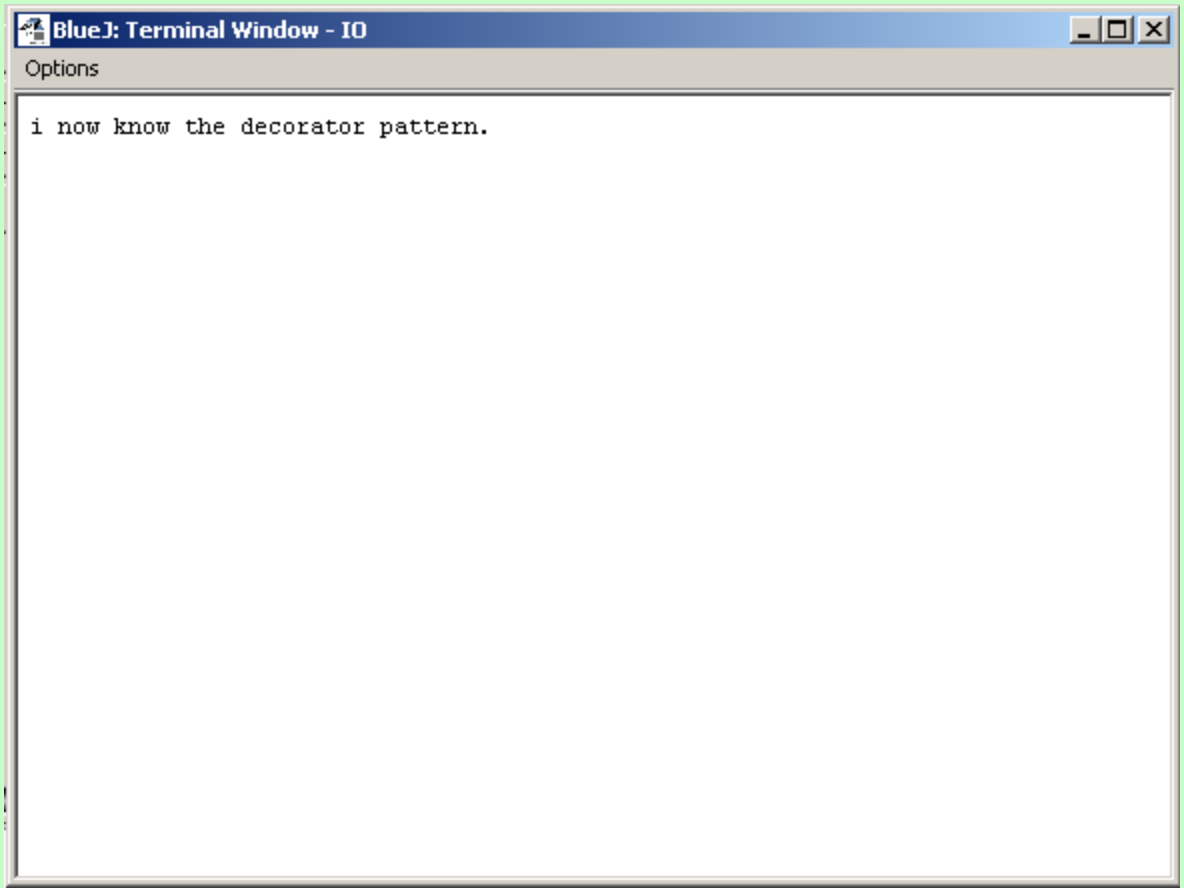
Implement 2 read() methods, taking a byte (or an array of bytes) and convert each byte to lowercase if it is an uppercase character

```
import java.io.*;

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 fileInputStream and decorate it, first with BufferedInputSteam and then LowercaseInputStream filter.

Executing project IO

- Test.txt contains “I now know the DECORATOR PATTERN.”



References

- Design Patterns: Elements of Reusable Object-Oriented Software By Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). Addison-Wesley. ISBN 0-201-63361-2.
- **Head First Design Patterns** By Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates
  First Edition October 2004
  ISBN 10: 0-596-00712-4
- http://www.uwosh.edu/faculty_staff/huen/262/f09/slides/15_Decorator_Pattern.ppt