Java IO. Part I. Files and Streams

Lecture 21

Java IO package

- Different sources and sinks of I/O:
  - files
  - console
  - network …
- You need to talk to them in a variety of ways:
  - sequential
  - random-access
  - buffered
  - binary
  - character
  - by lines …
- Result: 12 interfaces, 50 classes, 16 exceptions

Main utilities of java.io

- File manipulation
- Stream manipulation
- Serializing objects

Manipulating files: The File class

- The **File** class can represent either the *name* of a particular file or the *names* of a set of files in a directory.
- For a set of files use the **list( )** method, which returns an array of **Strings**.

Listing all files (and directories) in a current directory

```
public static void main(String[] args)
{
        File path = new File(“.");
        String[] list;

        list = path.list();

        Arrays.sort(list, String.CASE_INSENSITIVE_ORDER);

        for(String dirItem : list)
                System.out.println(dirItem);
}
```

Current directory

comparator

Listing all .java files

```
public static void main(String[] args){
        File path = new File("../");
        String[] list;
        list = path.list(new DirFilter(“.*\.java”));
}
```

Parent directory

FilenameFilter interface

```
class DirFilter implements FilenameFilter
{
        private Pattern pattern;
        public DirFilter(String regex) {
                pattern = Pattern.compile(regex);
        }

        public boolean accept (File dir, String name) {
                return pattern.matcher(name).matches();
        }
}
```

Parent directory → dir

Name of the file to be accepted or rejected → name

Strategy design pattern example

```
public static void main(String[] args){
        File path = new File("../");
        String[] list;
        list = path.list(new DirFilter(".*\.java"));
}
```

Matching strategy: part of a list() implementation

Digression: Regular expressions

- Regular expressions allow to specify, programmatically, complex patterns of text that can be discovered in an input string.
- A regular expression is a way to describe strings in general terms, so that you can say, "If a string has these things in it, then it matches what I’m looking for."
- A complete list of constructs for building regular expressions can be found in the documentation for the **Pattern** class in package **java.util.regex**.

Regular expression example 1

Starts with D

D.*\.java

Regular expression example 1

any character

repeated 0 or more times

D.*\.java

Regular expression example 1

Actual dot (with escape sequence)

D.*\.java

Regular expression example 2

Starts with g OR c

[gc].*

Regular expression example 3

[rR]**udolph**

[rR][aeiou][a-z]**ol**.*

R.*

- Which of these will match ***Rudolph***?

How to use java regular expressions

- Import **java.util.regex**
- Compile a regular expression by using the **static Pattern.compile( )** method. This produces a **Pattern** object based on its **String** argument.
- Use the **Pattern** by calling its **matcher( )** method, passing the string that you want to search. The **matcher( )** method produces a **Matcher** object

Pattern p = Pattern.compile(regexpression);

Matcher m = p.matcher(inputstring);

- Call methods of Matcher:

m.matches()   //boolean

Info about the file

```
public static void printFileData(File f) {
        System.out.println(
        "Absolute path: " + f.getAbsolutePath() +
        "\n Can read: " + f.canRead() +
        "\n Can write: " + f.canWrite() +
        "\n getName: " + f.getName() +
        "\n getParent: " + f.getParent() +
        "\n getPath: " + f.getPath() +
        "\n length: " + f.length() +
        "\n lastModified: " + f.lastModified());

        if(f.isFile())
                System.out.println("It's a file");
        else if(f.isDirectory())
                System.out.println("It's a directory");
}
```

```
created text2
Absolute path:
C:\Users\MGbarsky\workspa
ce\ioexamples\text2
 Can read: true
 Can write: true
 getName: text2
 getParent: null
 getPath: text2
 length: 0
 lastModified:
1352190726996
It's a directory
```

We can use the File class to rename,

File old = **new File(args[1]), rname = new File(args[2]);**

old.**renameTo**(rname);

We can use the File class to rename, delete

```
File f = new File(args[1]);
if(f.exists()) {
        System.out.println("deleting..." + f);
        f.delete();
}
```

We can use the File class to rename, delete, or create new directories

File f = **new File(args[1]);**

f.**mkdirs**();

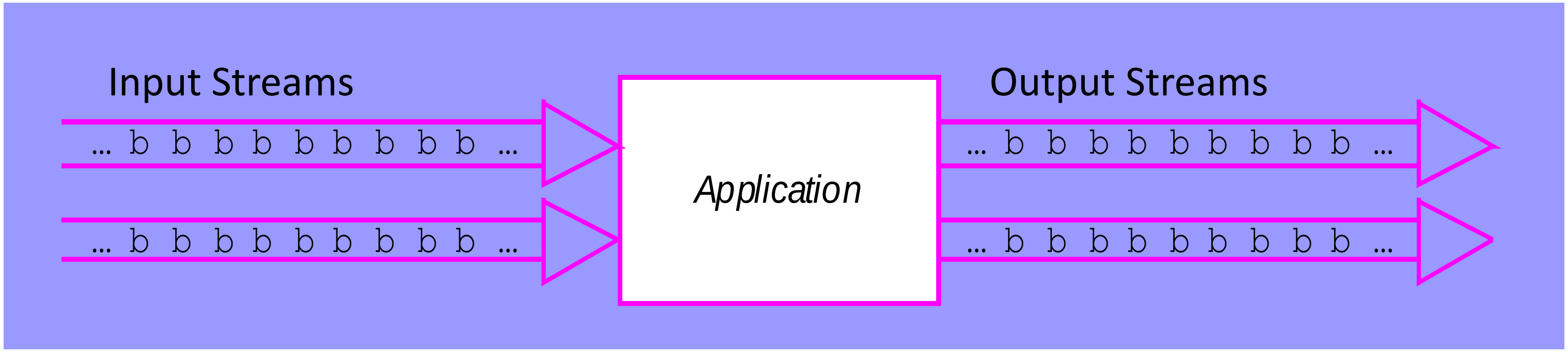
System.*out.println("created " + f);*

Main utilities of java.io

- File manipulation
- Stream manipulation
- Serializing objects

Input and output streams

- The abstraction of a *stream* represents any data source or sink as an object capable of producing or receiving pieces of data.
- The stream hides the details of what happens to the data inside the actual I/O device

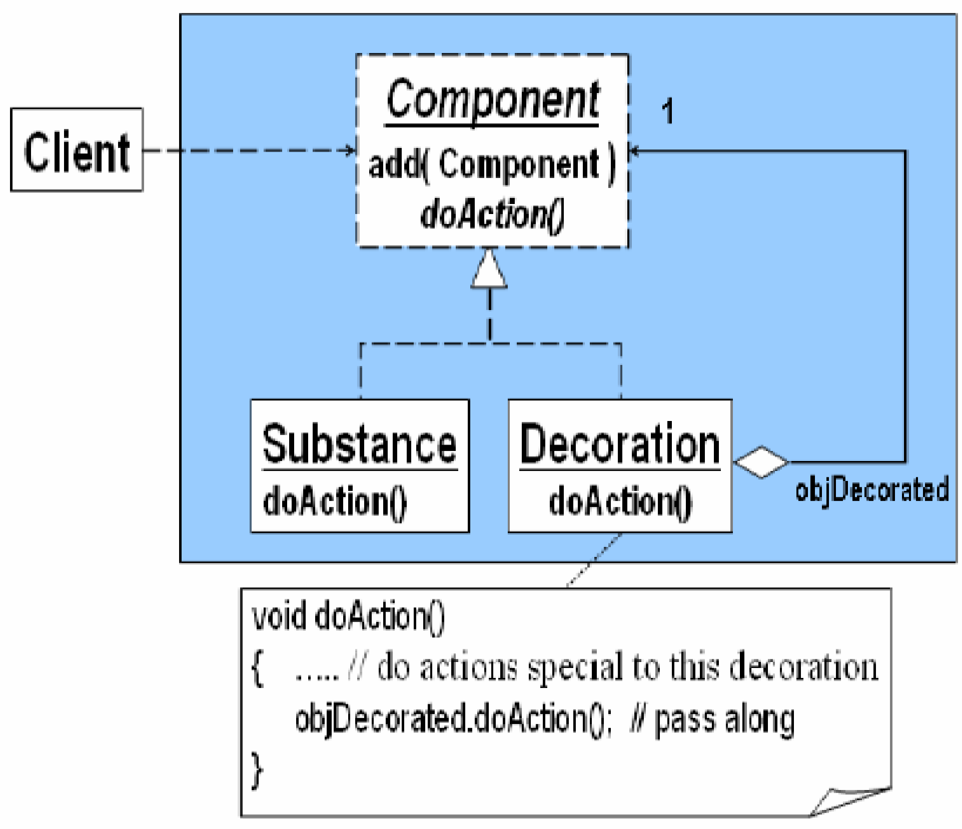


Digression: Decorator design pattern

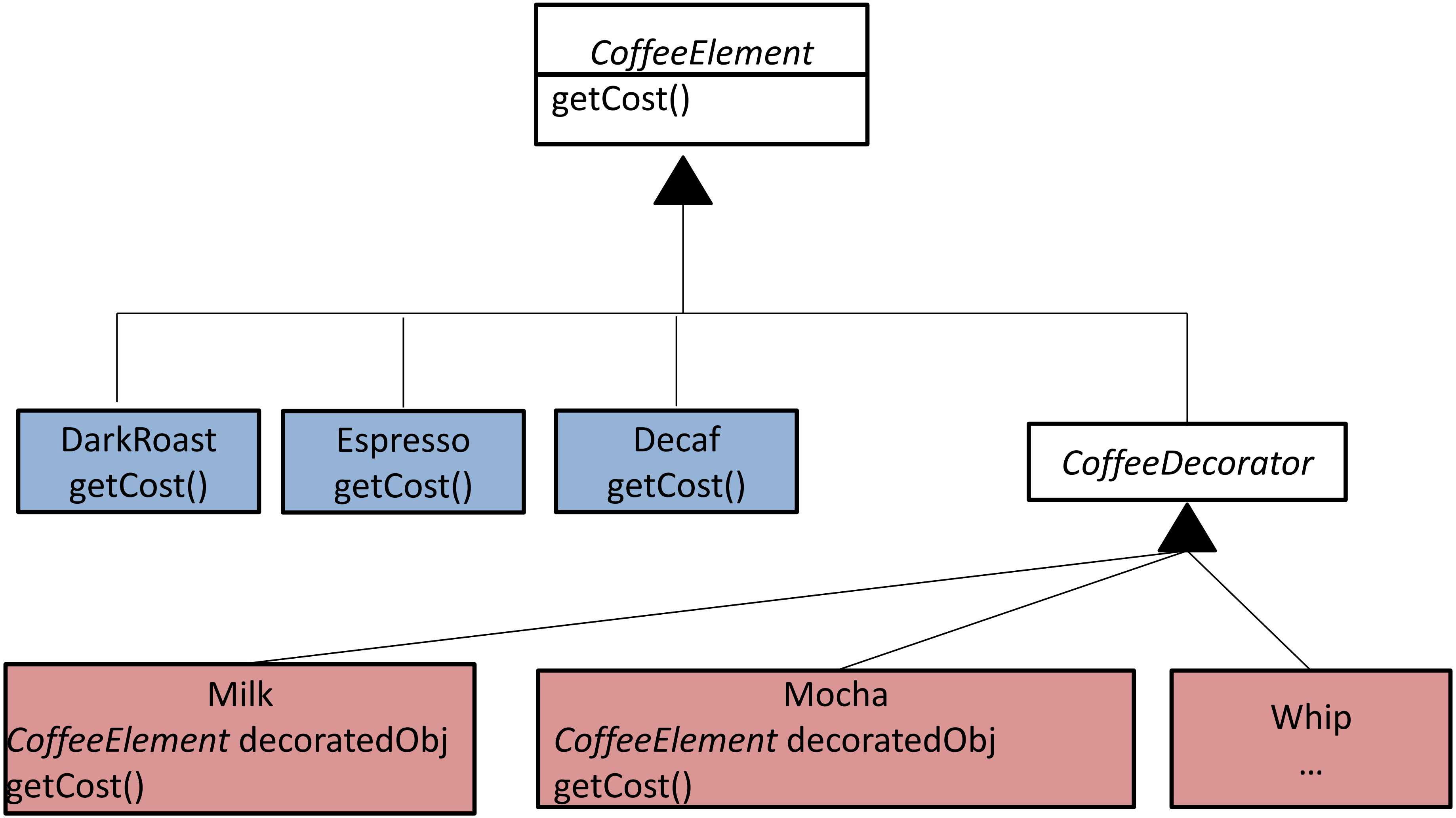
- Design Purpose: the use of layered objects to dynamically and transparently add responsibilities to individual objects
- Design Pattern Summary: provides a nested linked list of objects, each encapsulating its own responsibility.

Decorator UML diagram

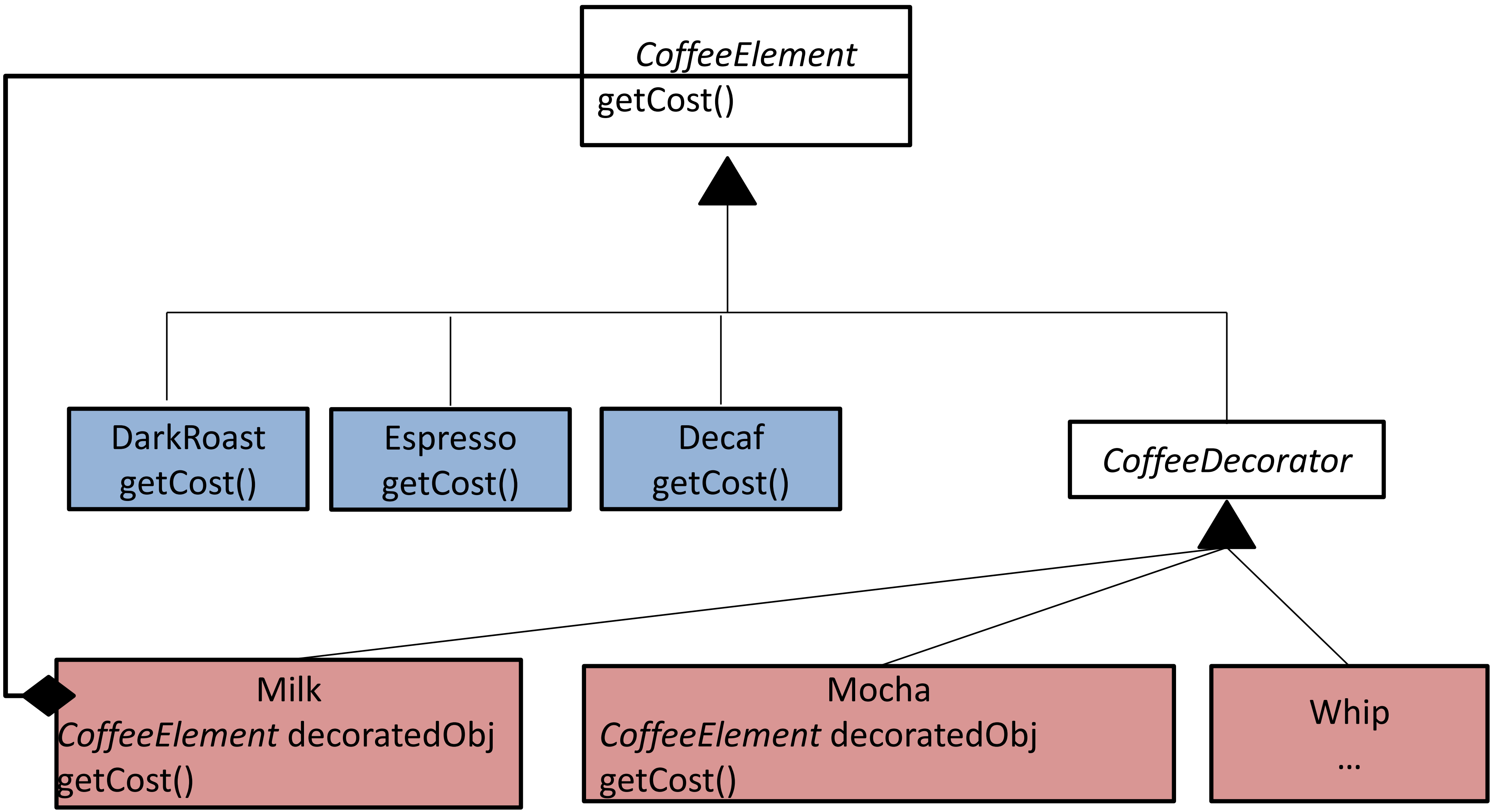
- The decorator pattern specifies that all objects that wrap around your initial object have the same interface.
- This makes the basic use of the decorators transparent—you send the same message to an object whether it has been decorated or not.



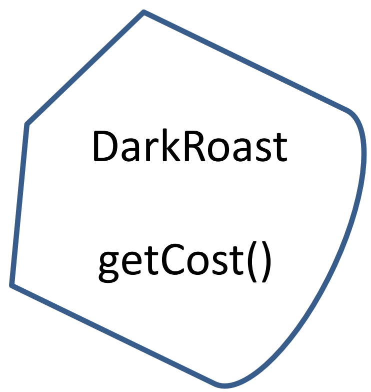
Coffee decorators example



Coffee decorators example



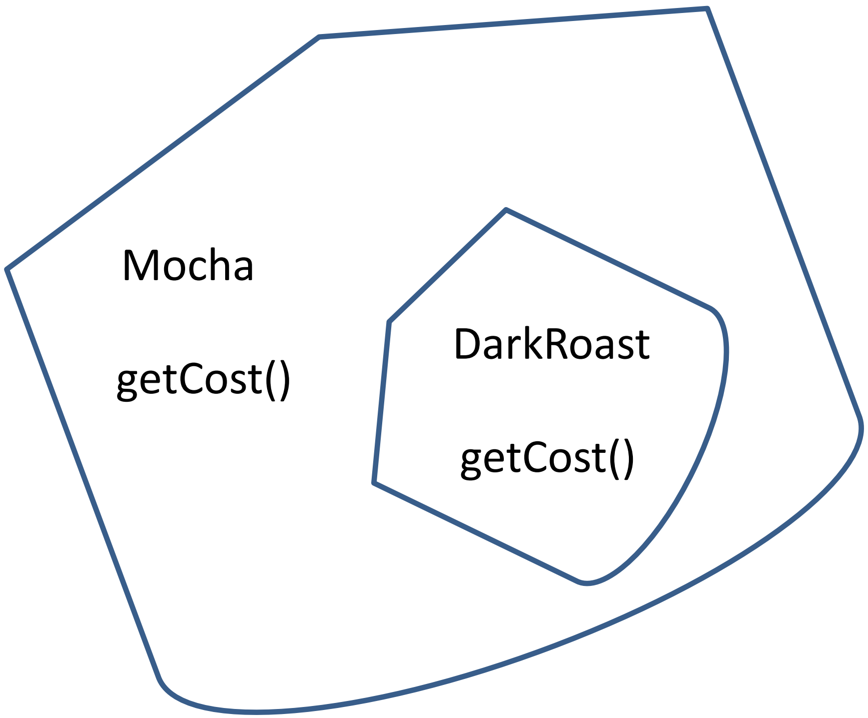
Nested decorators (1/3)



First, create a core coffee element:

CoffeeElement dark=new DarkRoast();

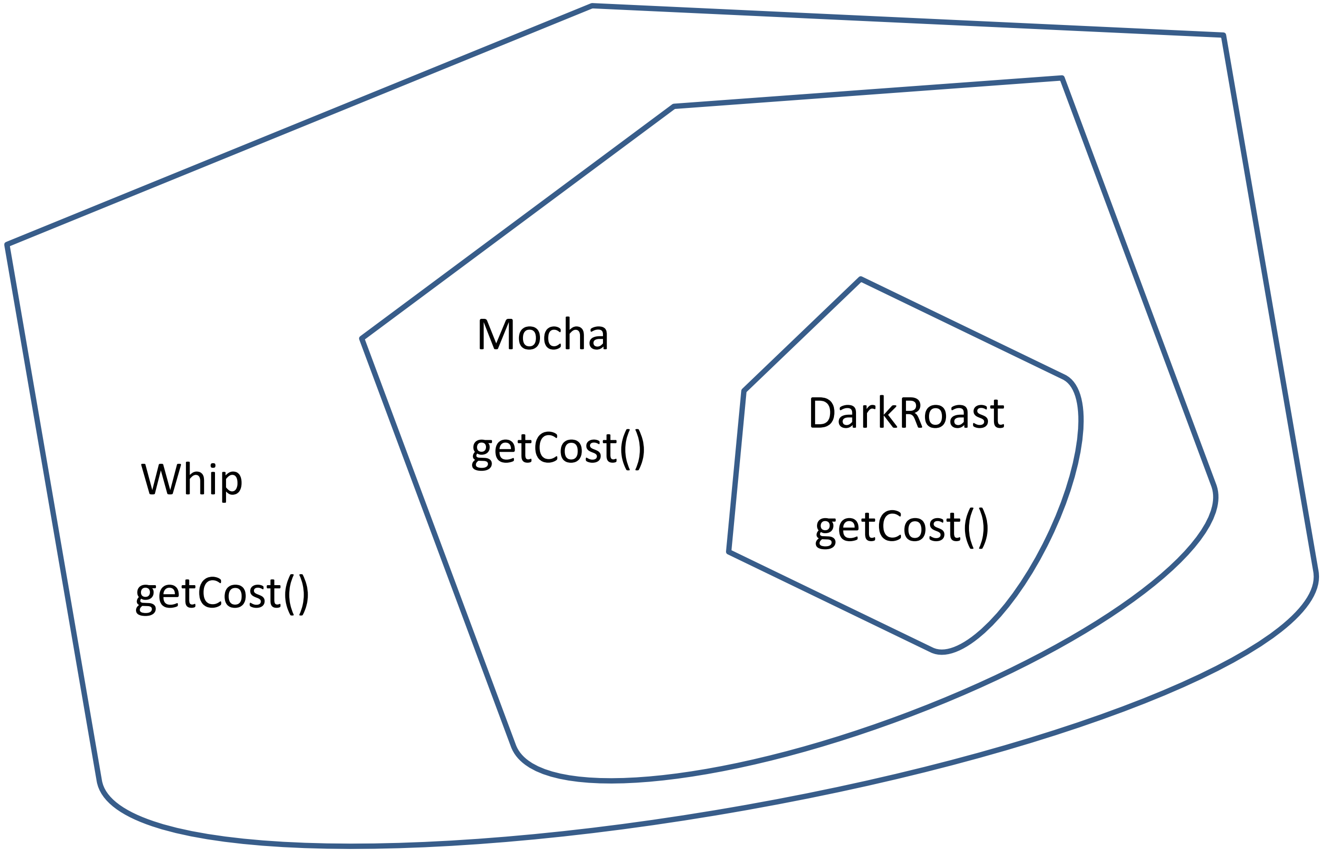
Nested decorators (2/3)



Wrap it with Mocha, pass DarkRoast to be decorated

Mocha whip=new Mocha (new DarkRoast());

Nested decorators



Wrap it with Whip, pass Mocha to be decorated

CoffeeElement whip=new Whip(new Mocha(new DarkRoast()));

Each CoffeeElement has getCost()

Subclass of CoffeeElement

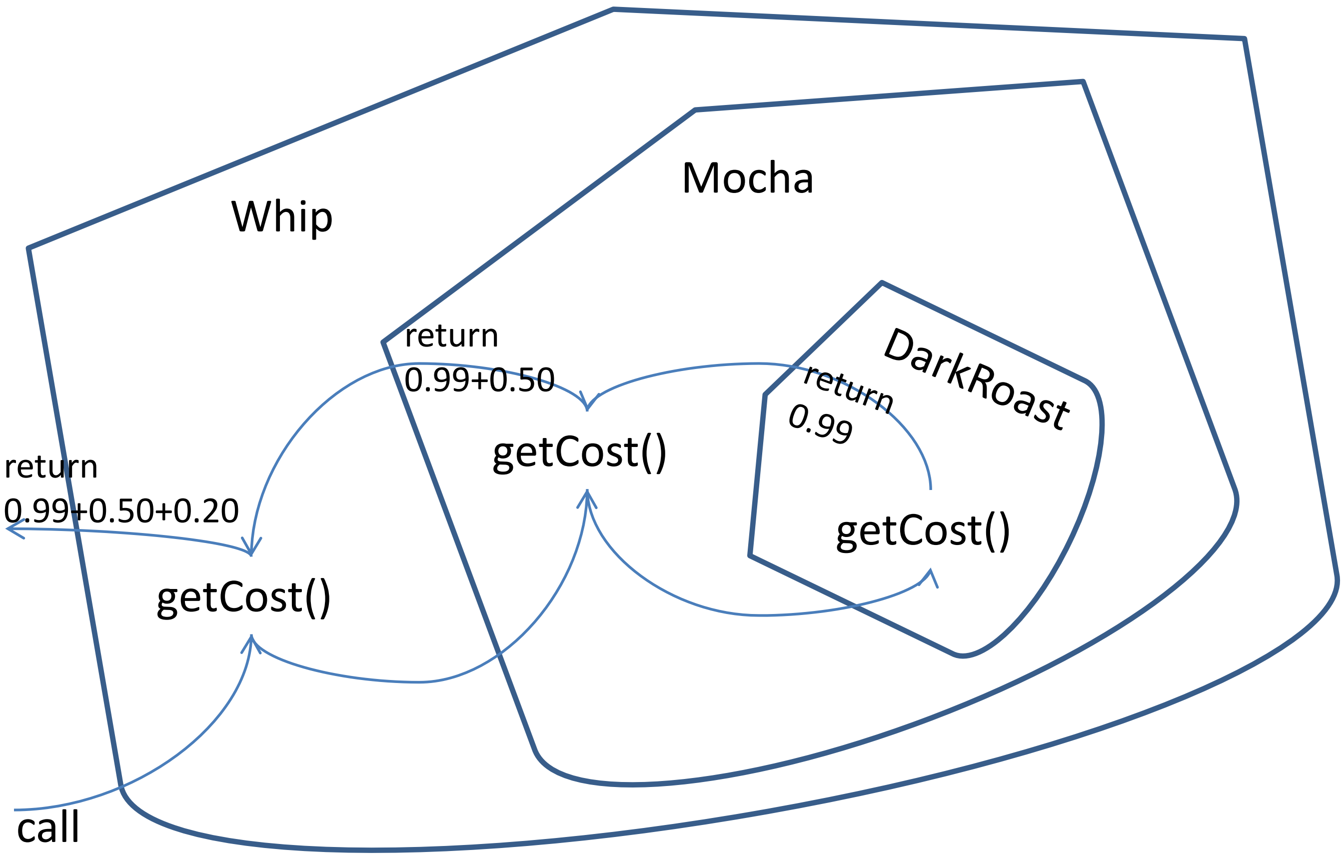
```
public class Whip extends CoffeeDecorator{
        CoffeeElement decoratedObj;

        public double getCost(){
                return 0.50 + decoratedObj.getCost();
        }
}
```

Its own cost

To compute the total beverage cost: call the getCost() of the outmost decorator

```
CoffeeElement beverage=new Whip(
                              new Mocha (
                                   new DarkRoast()));
cost=beverage.getCost();
```

This recursively adds costs of all elements



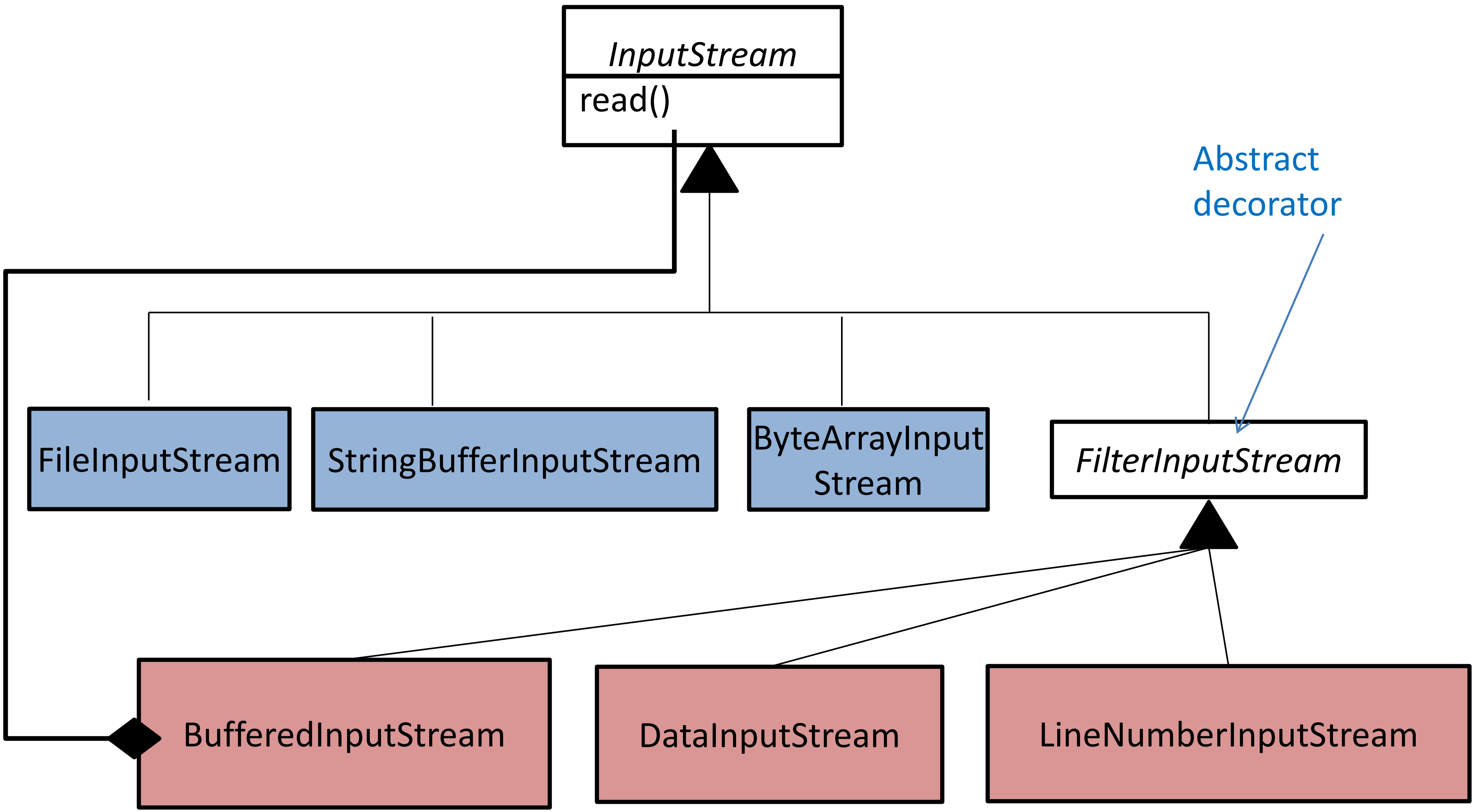
Favor composition over inheritance

- When we compose a decorator with a component, we are adding new behavior. We are acquiring new behavior not by inheriting it from a superclass, but by composing objects together.
- Because we are using object composition, we get a lot more flexibility about how to mix and match condiments and beverages.

The decorator design pattern: summary

- We can create endless combinations from basic decorators into an object with complex behavior
- We are changing the structure (and the functionality) of an object
- We can write new decorators to enable new behavior, and we do not need to change the old classes
- This demonstrates "**open for extension, closed for modification**" design principle

Back to java.io: Input Streams Inheritance tree



There are four main inheritance hierarchies with decorators

Read-write byte streams

- InputStream
- OutputStream

Read-write character streams (Unicode, 16 bit)

- Reader
- Writer

All mirror the Decorator pattern design

Abstract Decorator class in java.io

- The classes that provide the decorator interface to control a particular **InputStream** or **OutputStream** are the **FilterInputStream** and **FilterOutputStream**
- They are derived from the base classes of the I/O library, **InputStream** and **OutputStream**, which is the key requirement of the decorator (so that it provides the common interface to all the objects that are being decorated).

Concrete decorators for InputStreams

The **FilterInputStream** **subclasses**:

- **DataInputStream** allows you to read different types of primitive data and **String** objects. (**readByte( )**, **readFloat( )**, etc.) (corresponds to **DataOutputStream for an output)**

The remaining classes modify the way an **InputStream** behaves internally:

- Whether it's buffered - **BufferedInputStream**
- if it keeps track of the lines it's reading **LineNumberInputStream**
- …

Concrete decorators for OutputStreams

- The complement to **DataInputStream** is **DataOutputStream** (**writeByte( )**, **writeFloat( )**, etc.)
- **PrintStream** prints primitive data types and **String** objects in a viewable format (**print( )** and **println( )** are overloaded to print all the various types).
- **BufferedOutputStream** tells the stream to use buffering so you don't get a physical write every time you write to the stream. You'll probably always want to use this when doing output.

Reader and Writer hierarchies: reading/writing 16-bit Unicode Strings

Since Unicode is used for internationalization (and Java's native **char** is 16-bit Unicode), the **Reader** and **Writer** hierarchies were added to support Unicode in all I/O operations.

Example 1. Reading input by lines

```
BufferedReader in = new BufferedReader(
                new FileReader("IOStreamDemo.java"));
String s, s2 = new String();

while((s = in.readLine())!= null)
        s2 += s + "\n";
in.close();
```

Example 2. Reading standard input

```
BufferedReader stdin = new BufferedReader(
    new InputStreamReader(System.in));
System.out.print("Enter a line:");
System.out.println(stdin.readLine());
```

Example 3. Input from memory

```
String s2="abc";
StringReader in2 = new StringReader(s2);
int c;

while((c = in2.read()) != -1)
    System.out.print((char)c);
```

Example 4. File output

```
try {
        BufferedReader in4 = new BufferedReader(
                                        new StringReader(s2));
        PrintWriter out1 = new PrintWriter(
                new BufferedWriter(
                        new FileWriter("IODemo.out")));

        int lineCount = 1;

        while((s = in4.readLine()) != null )
                out1.println(lineCount++ + ": " + s);
        out1.close();
  } catch(EOFException e) {
        System.err.println("End of stream");
  }
```

Example 5. Storing and recovering data

```
DataOutputStream out2 = new DataOutputStream(
            new BufferedOutputStream(
                        new FileOutputStream("Data.txt")));
out2.writeDouble(3.14159);
out2.writeUTF("That was pi");
out2.close();

DataInputStream in5 = new DataInputStream(
            new BufferedInputStream(
                        new FileInputStream("Data.txt")));

// Must use DataInputStream for data:
    System.out.println(in5.readDouble());
    // Only readUTF() will recover the Java-UTF String properly:
    System.out.println(in5.readUTF());
```

Example 6. Reading and writing to standard input/output

```
BufferedReader in = new BufferedReader(
        new InputStreamReader(System.in));

    String s;
    while((s = in.readLine()) != null && s.length() != 0)
        System.out.println(s);
    // An empty line or Ctrl-Z terminates the program
    }
}
```

Example 7. Redirect standard output

```
PrintStream console = System.out;    //set console to restore later

BufferedInputStream in = new BufferedInputStream(
        new FileInputStream("Redirecting.java"));

PrintStream out = new PrintStream(
        new BufferedOutputStream(
                new FileOutputStream("test.out")));

System.setIn(in);
System.setOut(out);
System.setErr(out);

BufferedReader br = new BufferedReader(
                new InputStreamReader(System.in));
String s;
while((s = br.readLine()) != null)
        System.out.println(s);
out.close(); // Remember this!

System.setOut(console);  //restore
```

Example 8. Our own new I/O decorator

```
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;
	}
}
```

Summary of stream manipulation

- Using the decorator design pattern, we can create a nested combination of Stream readers/writers suitable for our needs
- We wrap the core elements, which extend directly from a top abstract class, with the decorator classes, which extend decorator abstract class (Filter abstract classes, for example)

Why decorators in java.io

- Decorators are often used when simple subclassing results in a large number of classes in order to satisfy every possible combination that is needed
- The Java I/O library requires many different combinations of features, and this is the justification for using the decorator pattern
- Decorators give you much more flexibility while you’re writing a program (since you can easily mix and match attributes), but they add complexity to your code.

Decorator pattern: downsides

- Designs often result in a large number of small classes that can be overwhelming to a client programmer.
- The reason that the Java I/O library is awkward to use is that you must create many classes—the "core" I/O type plus all the decorators—in order to get the single I/O object that you want
- But now that you know how Decorator works, you can keep things in perspective and use wrapping to get the behavior you want.

Main utilities of java.io

- File manipulation
- Writing and reading of Streams
- Serializing objects