

***Software
Technology
Group***

TU Darmstadt | FB Informatik

Software Engineering Design

5. Design Patterns: Decorator

Prof. Dr. Mira Mezini

Dipl.-Inform. Christoph Bockisch

Dipl.-Ing. Michael Haupt

Document Editor: Design Issue

- **embellishing the user interface...**
 - add borders around text editing areas
 - add scroll bars that let the user view different parts of the text
 - ...
 - combinations thereof
- from a programming point of view, embellishing the user interface involves extending existing code
 - a bordered text area is a special kind of text area
- **constraints**
 - no global embellishment: individual objects
 - flexibility
 - transparency

Embellishing the UI

Use Inheritance?

- assume we have a class named **VisualComponent**
- could add a border to **VisualComponent** by subclassing it to yield **BorderedVisualComponent**
- could add a scrolling interface in the same way to yield a **ScrollableVisualComponent**
- if we want scroll bar and border, we produce **BorderedScrollableVisualComponent**, ...

Problems with Inheritance

- explosion of number of subclasses
- not dynamic: object life-time decisions

Embellishing the UI

Object Composition

- offers a potentially more flexible extension mechanism
 - make embellishments themselves objects (say, instances of **Border**, **Scroller**)
 - compose components with embellishment objects

Direction of Composition Relationships

- **embellishments** contain visual components
 - natural approach
 - embellishment code is kept in the embellishment classes alone
- **visual components** contain embellishments
 - entails modifications in visual component classes
 - visual components need to be aware of embellishments

Embellishing the UI

How Does the Design Look?

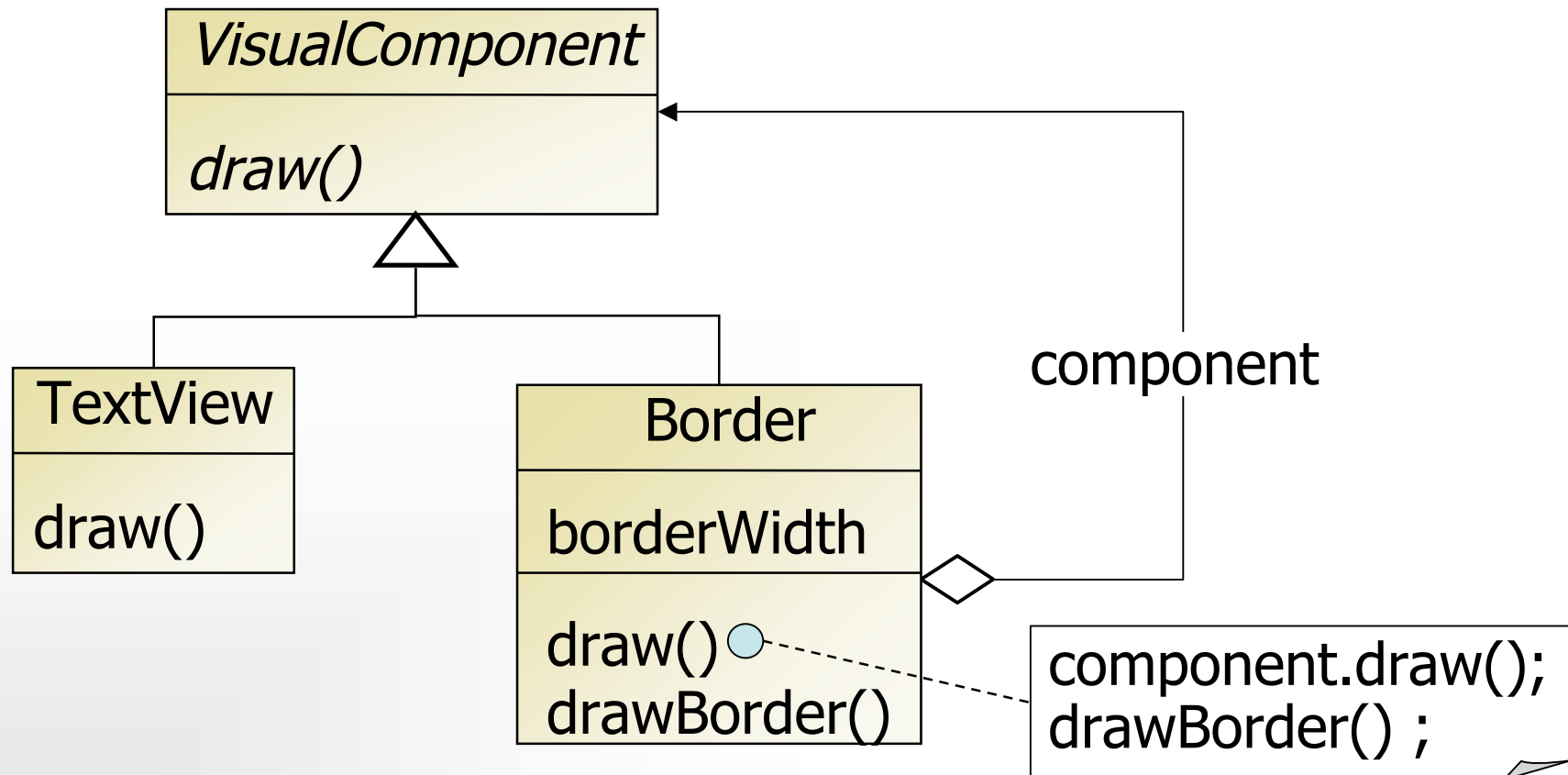
- **Border** objects contain **VisualComponent** objects
- **Borders** are themselves **VisualComponents**
- clients should treat all **VisualComponents** uniformly
 - tell a plain element to draw itself: no embellishments
 - if element is embellished, there should be no difference to client

Consequences

- implication: **Border**'s interface matches that of **VisualComponent**
- solutions to guarantee this
 - subclass **Border** from **VisualComponent** (C++)
 - let **Border** implement the **VisualComponent** interface (Java)

Embellishing the UI

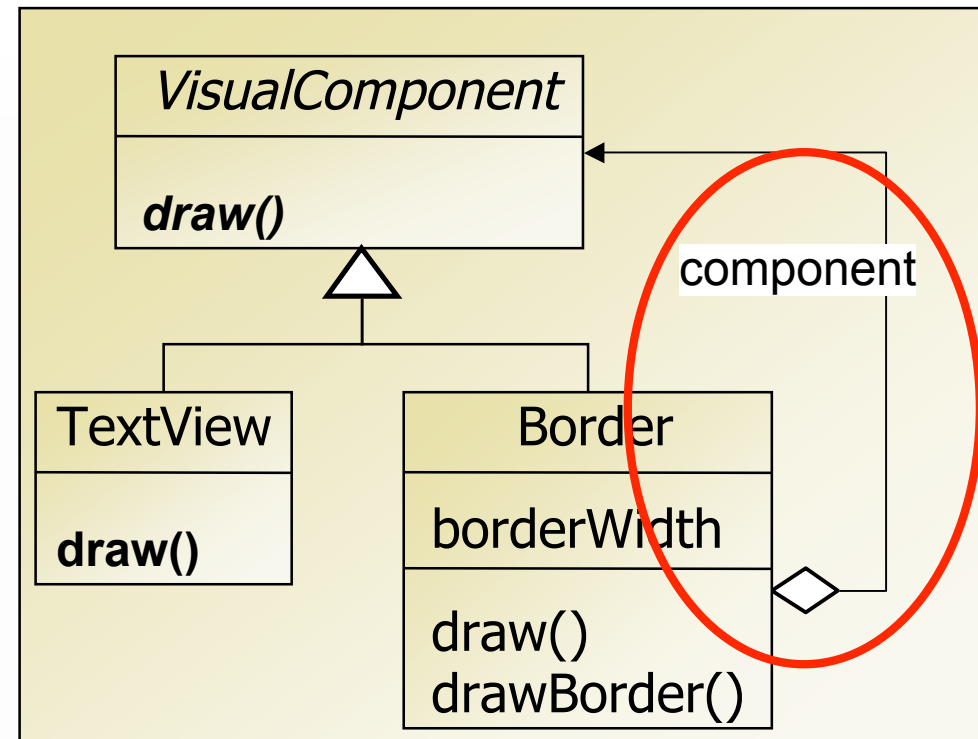
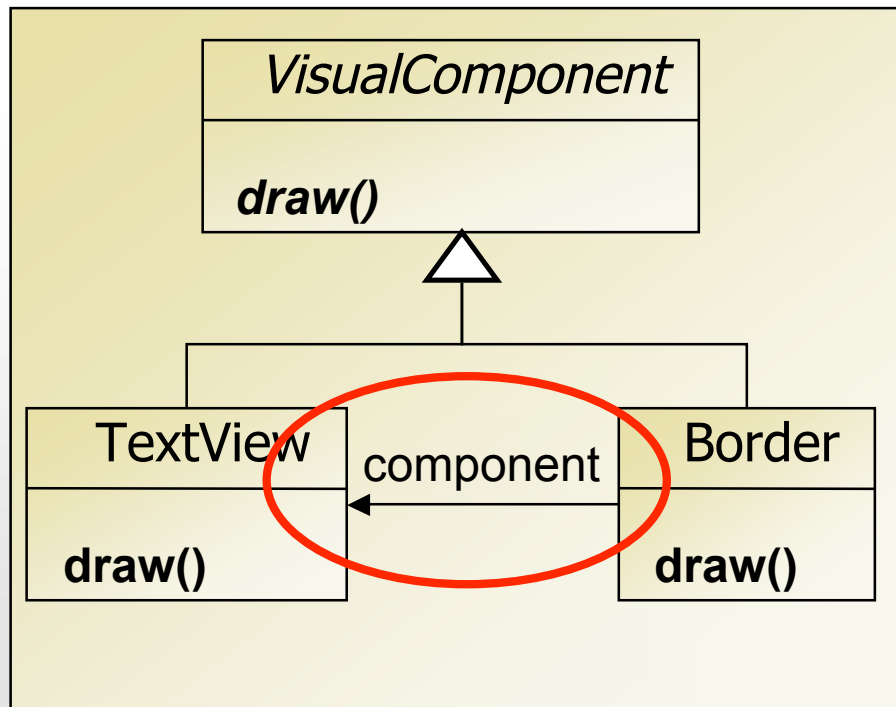
Object Composition: How does the Design Look?



Embellishing the UI

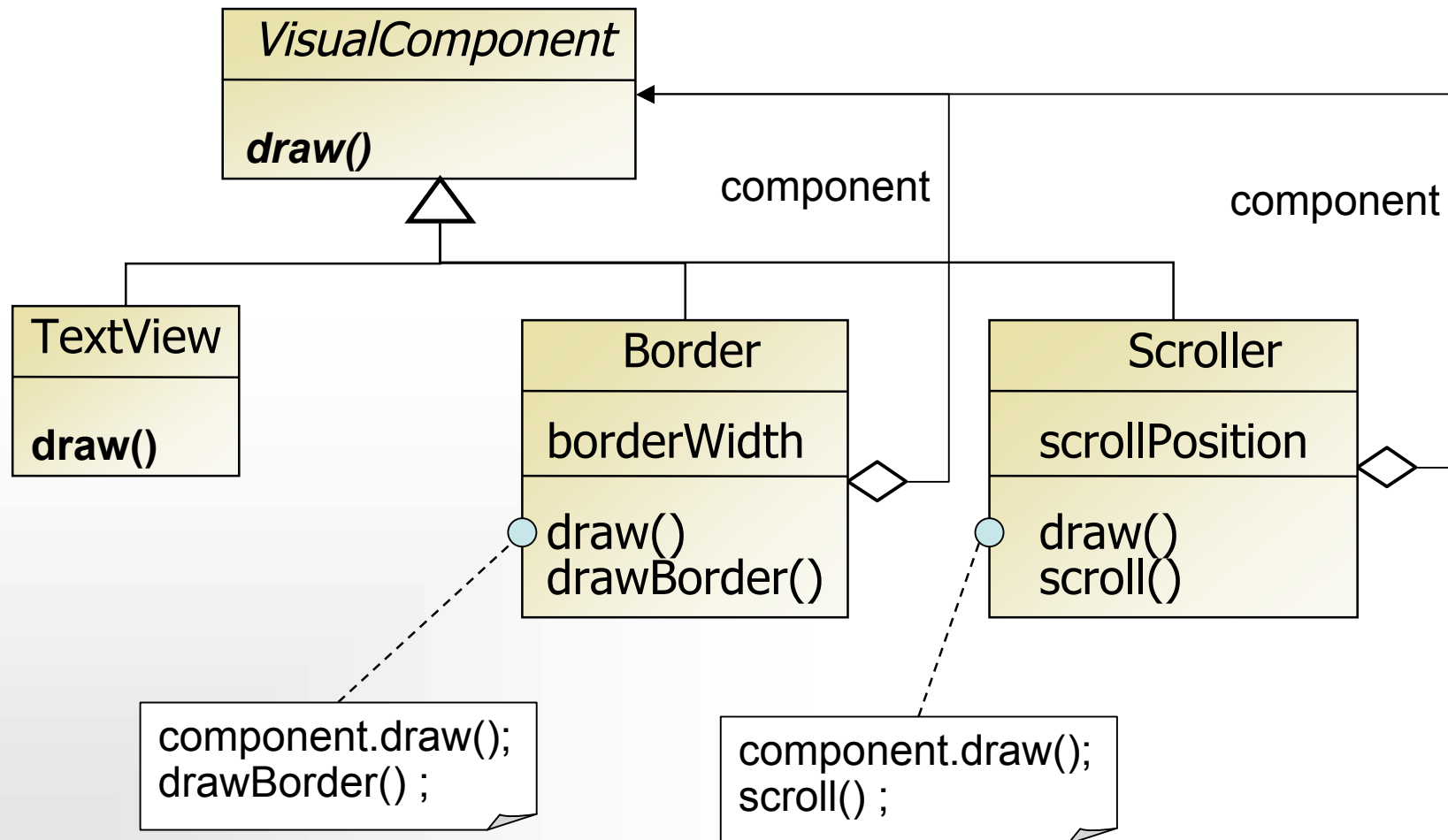
Transparent Enclosure

- single-child composition
- substitutability
- transparency
- **augmentation** of additional state and behaviour

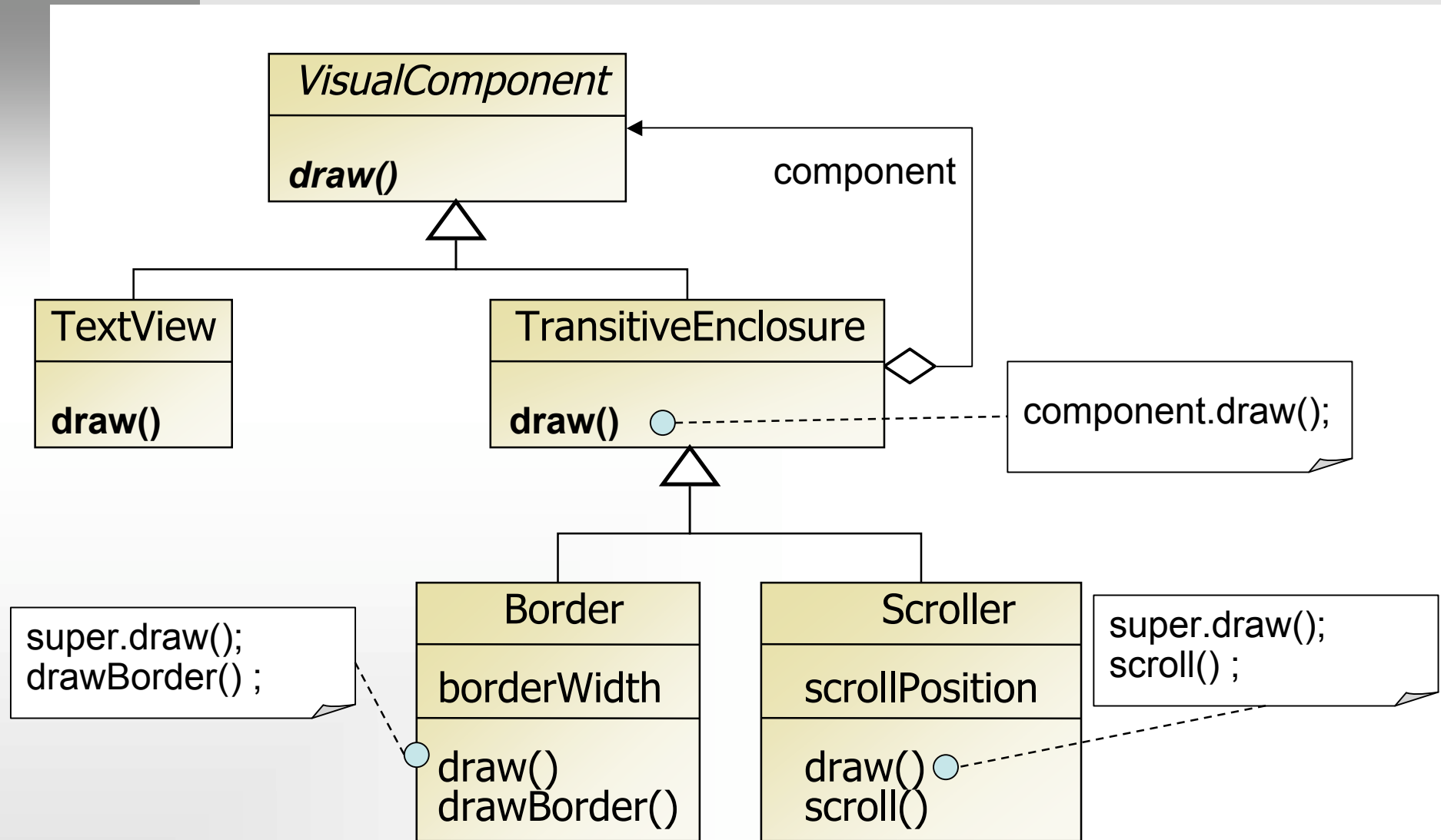


Embellishing the UI

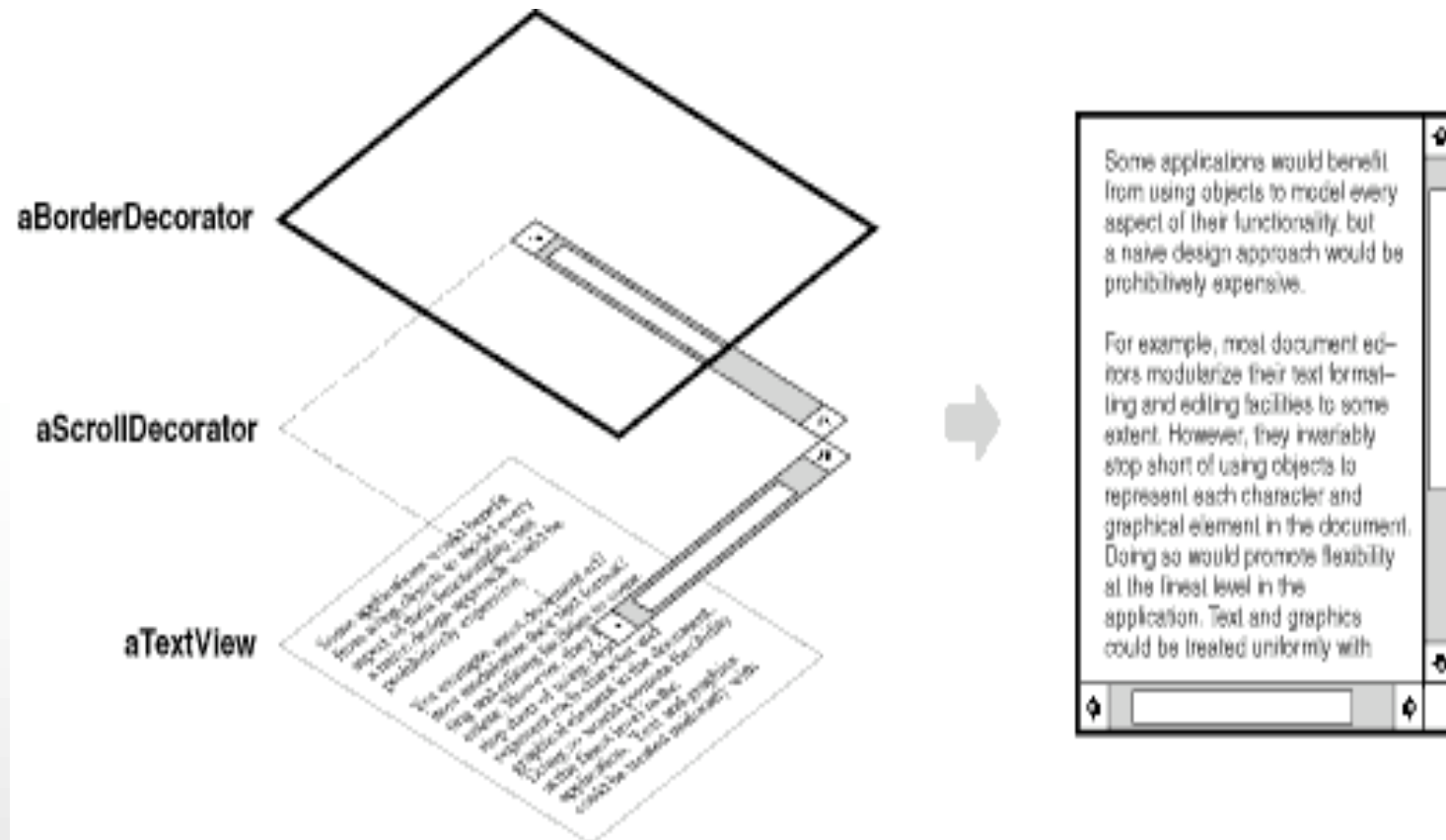
What do Scroller and Border Look Like?



Border and Scroller as Decorators



Border and Scroller as Decorators



Border and Scroller as Decorators

```
class Window {  
    VisualComp content;  
    public Window() { }  
    public void setContent(VisualComp vc) {  
        content = vc;  
    }  
}
```

```
interface VisualComp {  
    public void draw();  
    public void resize();  
}
```

Border and Scroller as Decorators

```
class TextView implements VisualComp {
    public TextView() { }
    public void draw() {
        System.out.println(
            "plain component drawing..."
        );
    }
    public void resize() {
        System.out.println(
            "plain component resizing..."
        );
    }
}
```

Border and Scroller as Decorators

```
class Decorator implements VisualComp {
    private VisualComp component;
    public Decorator(VisualComp vc) {
        component = vc;
    }
    public void draw() {
        component.draw();
    }
    public void resize() {
        component.resize();
    }
}
```

Border and Scroller as Decorators

```
class ScrollDecorator extends Decorator {
    private int scrollSize;
    public ScrollDecorator(
        VisualComp vc, int size
    ) {
        super(vc);
        scrollSize = size;
    }
    public void draw() {
        super.draw();
        scroll();
    }
    public void scroll() {
        System.out.println("Scrolling...");
    }
}
```

Border and Scroller as Decorators

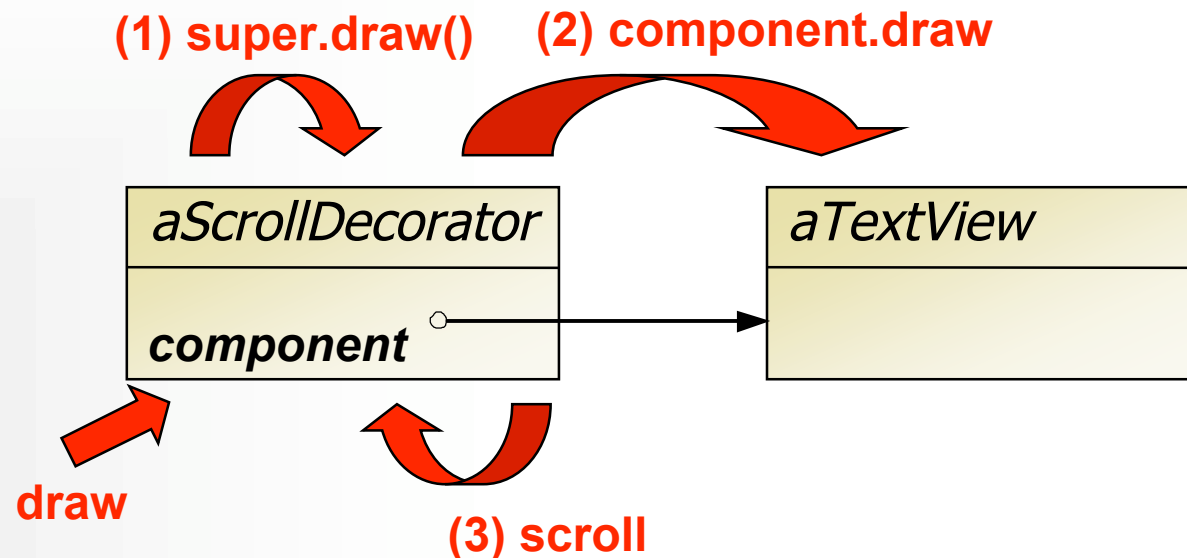
```
class BorderDecorator extends Decorator {
    private int width;
    public BorderDecorator(
        VisualComp vc, int width
    ) {
        super(vc);
        this.width = width;
    }
    public void draw() {
        super.draw();
        drawBorder();
    }
    public void drawBorder() {
        System.out.println("Drawing border...");
    }
}
```

Border and Scroller as Decorators

```
class Client {  
    public static void main(String[] args) {  
        Window window = new Window();  
        TextView tv = new TextView();  
        window.setContent(  
            new BorderDecorator(  
                new ScrollerDecorator(tv, 500),  
                1  
            )  
        );  
    }  
}
```

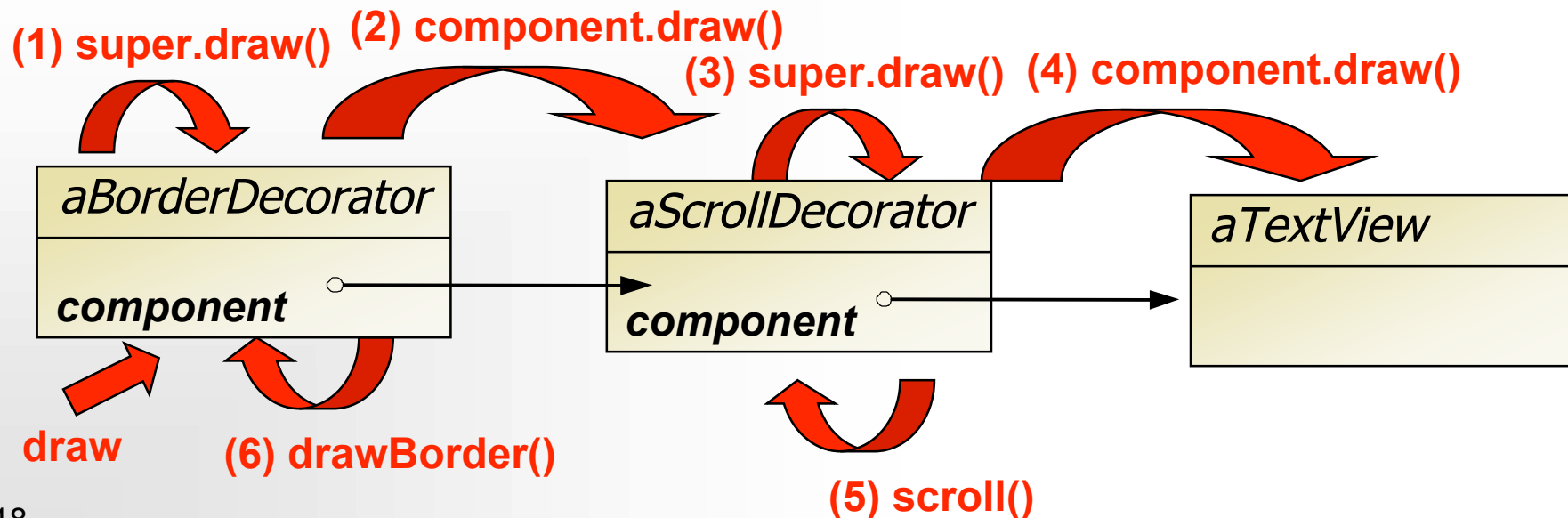
Border and Scroller as Decorators

```
VisualComp vComp = new TextView(text) ;  
vComp.draw() ;  
...  
vComp = new ScrollDecorator(vComp) ;  
vComp.draw() ;  
...  
vComp = new BorderDecorator(vComp) ;  
vComp.draw() ;
```

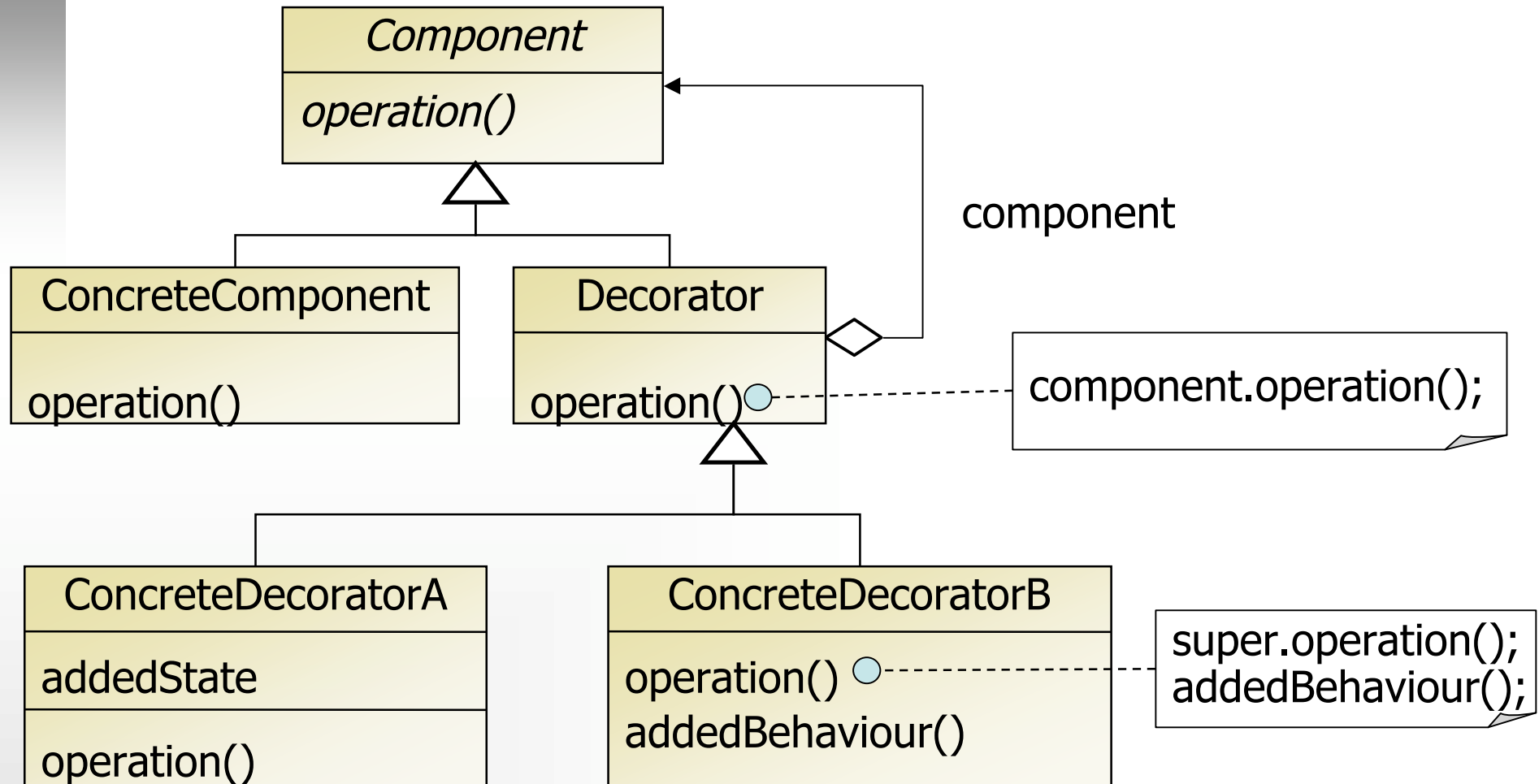


Border and Scroller as Decorators

```
VisualComp vComp = new TextView(text) ;  
vComp.draw() ;  
  
...  
vComp = new ScrollDecorator(vComp) ;  
vComp.draw() ;  
  
...  
vComp = new BorderDecorator(vComp) ;  
vComp.draw() ;
```

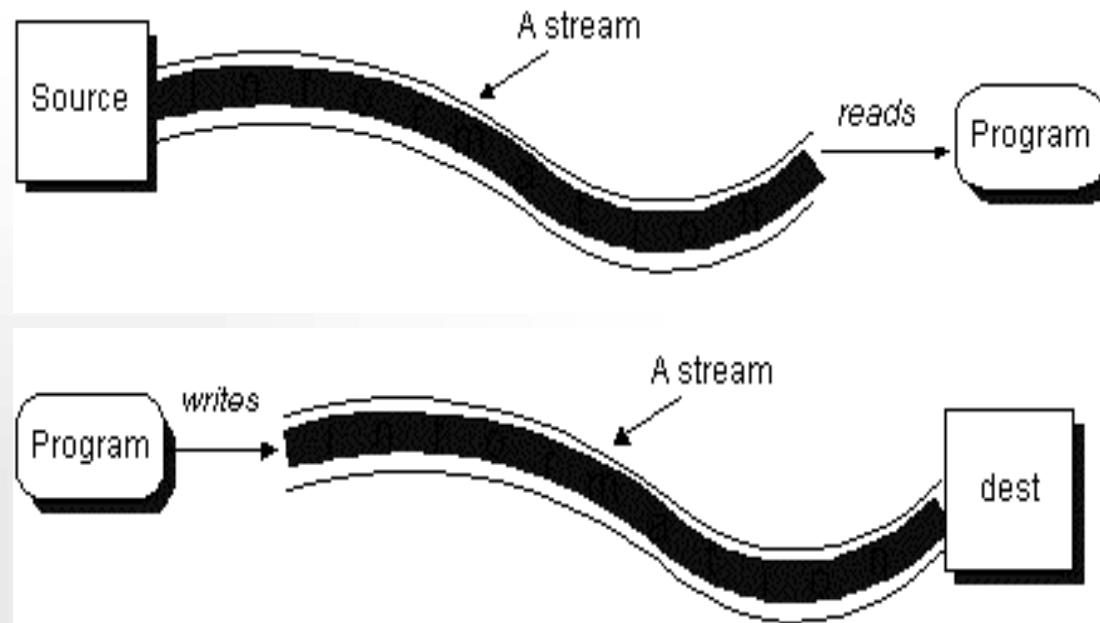


Decorator Structure



Decorator Case Study: java.io

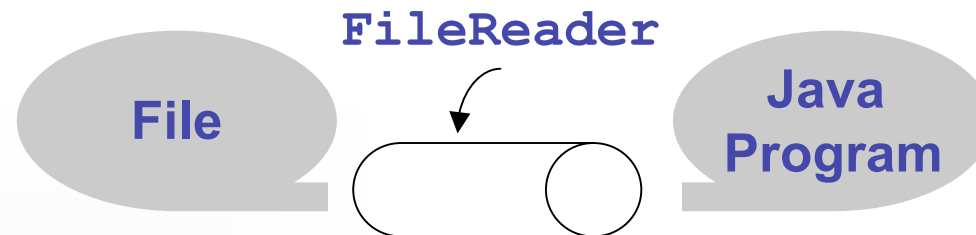
- Java **IO-Stream**: an **ordered sequence of data items** either **read** from some source **or written** to some destination, possibly with **processing along the way**
- two abstract root superclasses: **InputStream**, **OutputStream**



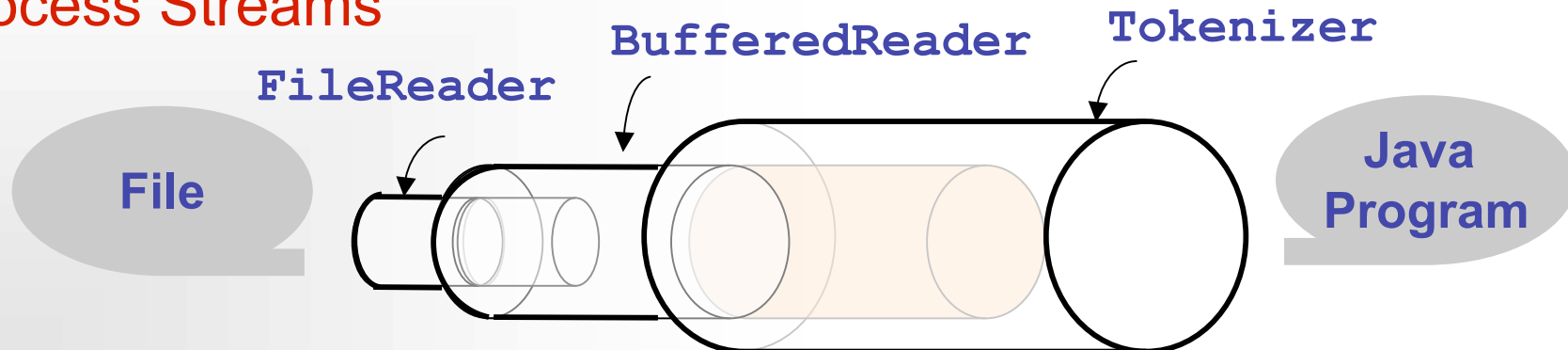
Decorator Case Study: java.io

- the external source/sink medium can itself be a stream
- goal: data can be processed on the way from program to sink, respectively from source to program

Data Source/Sink Streams

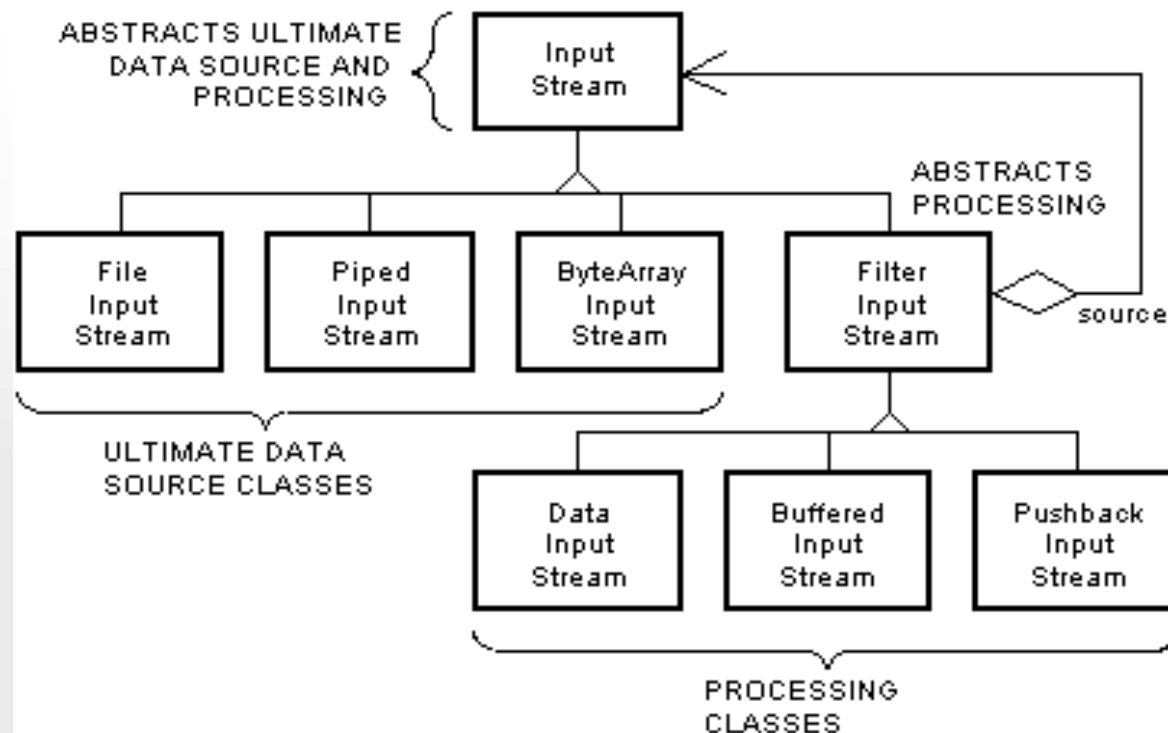


Process Streams

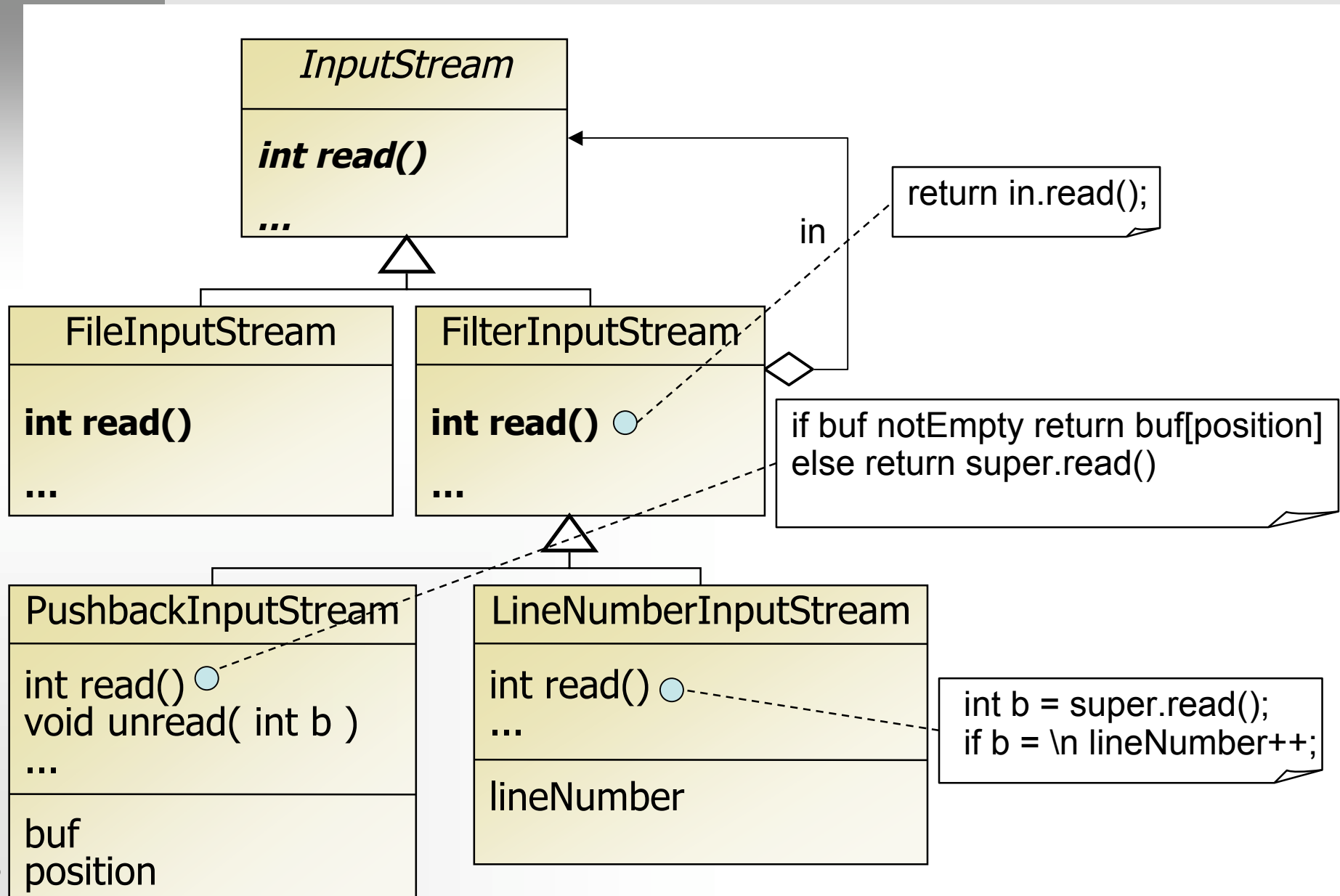


Decorator Case Study: java.io

- `java.io` abstracts various encountered sources, destinations, and processing algorithms
- Java programs operate on stream objects independently of
 - ultimate data source / destination
 - shape of data
- achieved by using the **Decorator pattern**



Decorator in java.io Package



Decorator in java.io Package

```
public abstract class InputStream {
    public int abstract read();
    public int read(byte b[ ]);
    public int read(byte b[ ], int off, int len);
    public long skip(long n);
    public int available();
    public void close();
    . . .
}
```

```
public class FilterInputStream extends InputStream {
    protected InputStream source;
    protected FilterInputStream(InputStream in) {
        source = in;
    }
    public int read() {
        return source.read();
    }
    . . .
}
```

Decorator in java.io Package

```
public class PushbackInputStream extends
FilterInputStream {
    protected int buffer = -1;
    public PushbackInputStream(InputStream in) {
        super(in);
    }
    public int read() {
        int r;
        if ( buffer != -1 ) {
            r = buffer;
            buffer = -1;
        }
        else { r = super.read(); }
        return r;
    }
    public void unread(int ch) {
        if ( buffer != -1 ) { // throw an exception ... }
        buffer = ch;
    }
    . . .
}
```

Decorator in java.io Package

... Processing ... Converting between (user-defined) data types and bytes ...

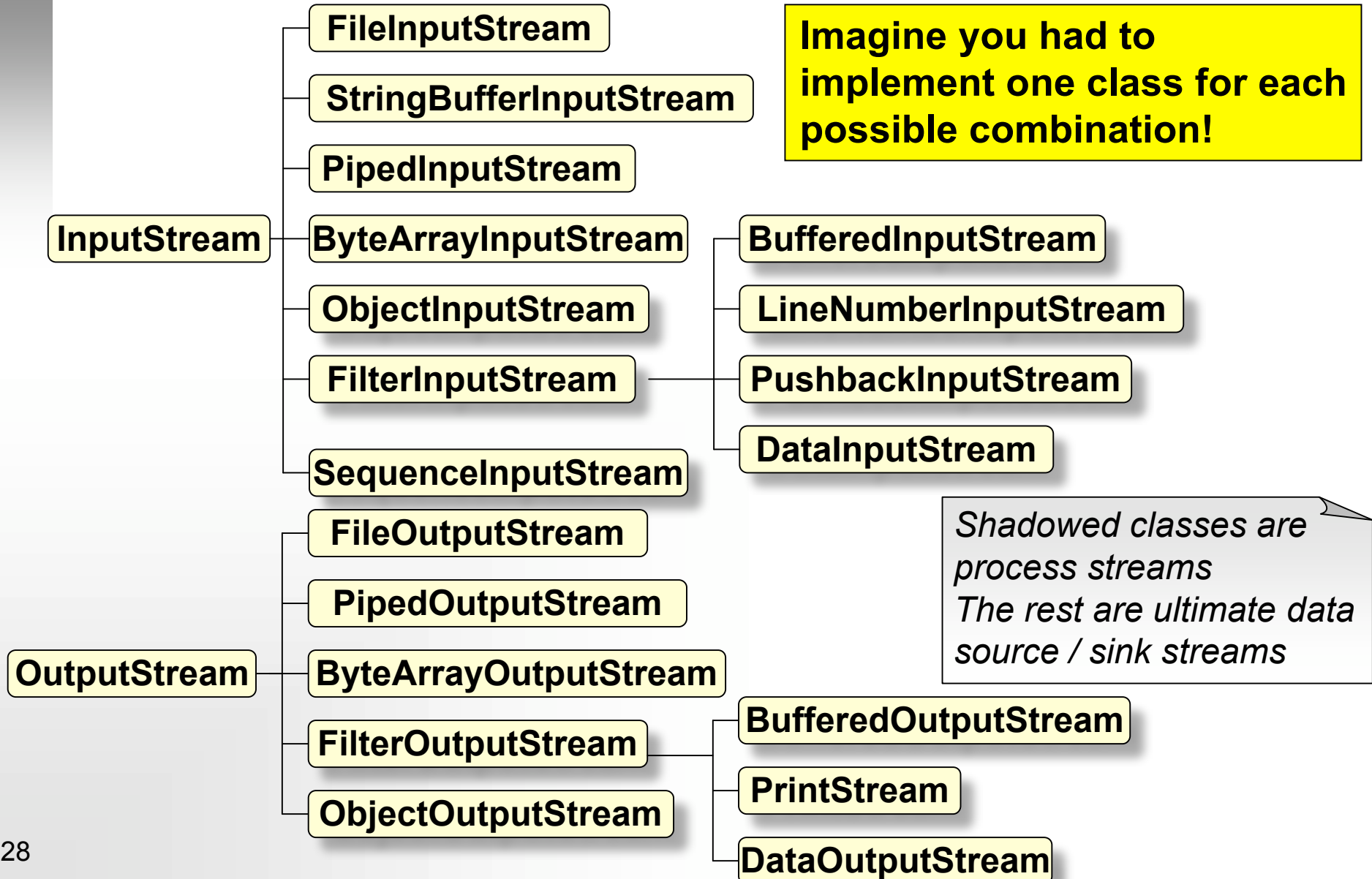
```
public class PrintStream extends FilterOutputStream {  
  
    public PrintStream(OutputStream out) { . . . }  
    public void print( Object o ) { . . . }  
    public void print( char c ) { . . . }  
    public void print( int i ) { . . . }  
    public void print( float f ) { . . . }  
    public void print( String s ) { . . . }  
  
    . . .  
    public void println( Object o ) { . . . }  
    public void println( char c ) { . . . }  
    public void println( int i ) { . . . }  
    public void println( float f ) { . . . }  
    public void println( String s ) { . . . }  
  
    . . .  
}
```

Decorator in java.io Package

. . . Processing . . . Reading bytes and returning data types to the program . . .

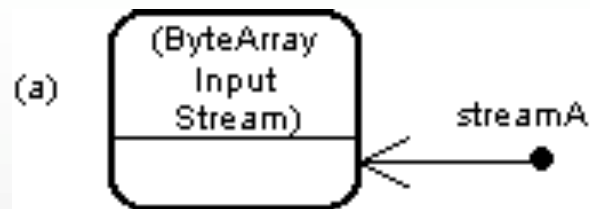
```
public class DataInputStream extends FilterInputStream
{
    public boolean readBoolean();
    public byte readByte();
    public char readChar();
    public int readInt();
    public String readLine();
    . . .
}
```

Decorator in java.io Package



Decorator in java.io Package

- this design is highly **run-time configurable**
- the decorator pattern generates constellations of objects that look like singly-linked lists and that **mix and match ultimate data sources and processing**
- arbitrary number of processing objects can decorate the ultimate data source



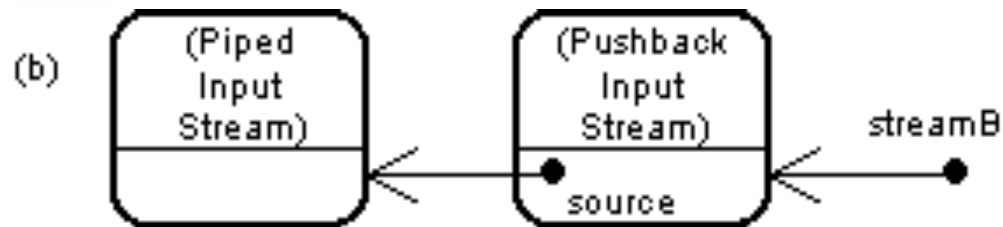
Constellation 1

- ultimate data source: in-core array
no additional processing

```
byte[ ] buffer = new byte[1024];  
// Fill buffer . . .  
InputStream streamA =  
    new ByteArrayInputStream(buffer);
```

Decorator in java.io Package

. . . constellations of objects that mix and match ultimate data sources and processing . . .



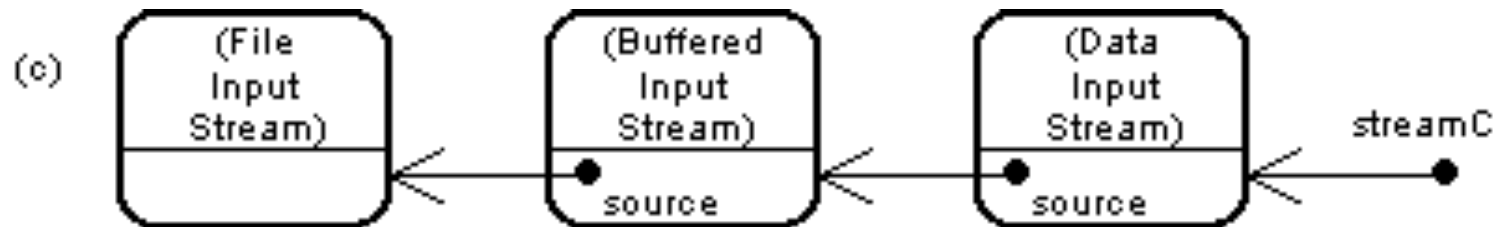
Constellation 2

- a separate thread as ultimate data source via
- a PipedInputStream object with additional single byte pushback buffer processing

```
PipedOutputStream out = new  
    PipedOutputStream(...);  
PipedInputStream in = new  
    PipedInputStream(out);  
InputStream streamB = new  
    PushbackInputStream(in);
```

Decorator in java.io Package

. . . constellations of objects that mix and match ultimate data sources and processing . . .



Constellation 3:

adds buffering and typed data interpretation to an unbuffered byte stream from a disk file.

```
FileInputStream fis =  
    new FileInputStream("somefile.txt");  
BufferedInputStream bis =  
    new BufferedInputStream(fis);  
DataInputStream streamC =  
    new DataInputStream(bis);
```

Decorator in java.io Package

. . . constellations of objects that mix and match ultimate data sources and processing . . .

Other sample processing might...

- uppercase a byte stream from across the network,
- count the number of lines in a stream originating from a file
- compress, encrypt, ...

Decorator in java.io Package

```
public class ReadCountInputStream
extends FilterInputStream {

    int reads = 0;
    public ReadCountInputStream(InputStream in) {
        super(in);
    }
    public int read() {
        reads++;
        return super.read();
    }
    . . .
}
```

Decorator Applicability

- need to add responsibilities to **individual objects**
 - dynamically and transparently, that is, without affecting other objects
 - responsibilities that can be withdrawn
- when extension by **subclassing is not practical**
 - large number of independent extensions are possible and would produce an explosion of subclasses to support every combination
 - a class definition may be hidden or otherwise unavailable for subclassing

Decorator Advantages

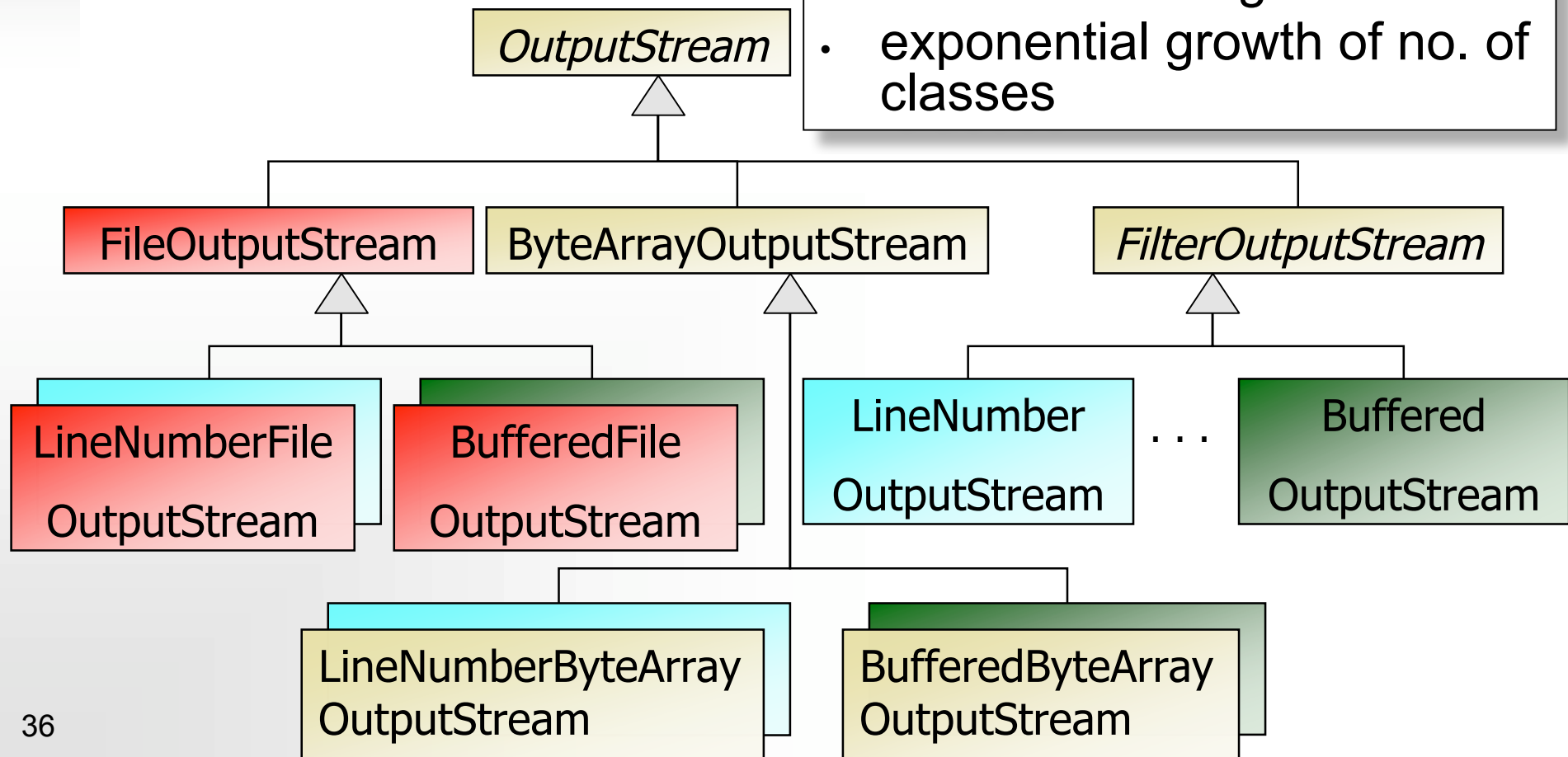
More Flexible than Static Inheritance

- Decorator
 - responsibilities can be added / removed at run-time by attaching and detaching them
 - providing different **Decorator** classes for a specific **Component** class lets you mix and match responsibilities
- inheritance
 - requires creating a new class for each additional responsibility (e.g., **BorderedScrollableTextView**, **BorderedTextView**)
 - the mix of responsibilities is statically fixed at object creation time
- easy to add a property twice
 - for example, to give a **TextView** a double border, simply attach two **BorderDecorators**
- inheriting from a **Border** class twice is error-prone at best

Inheritance Disadvantages

Single Inheritance

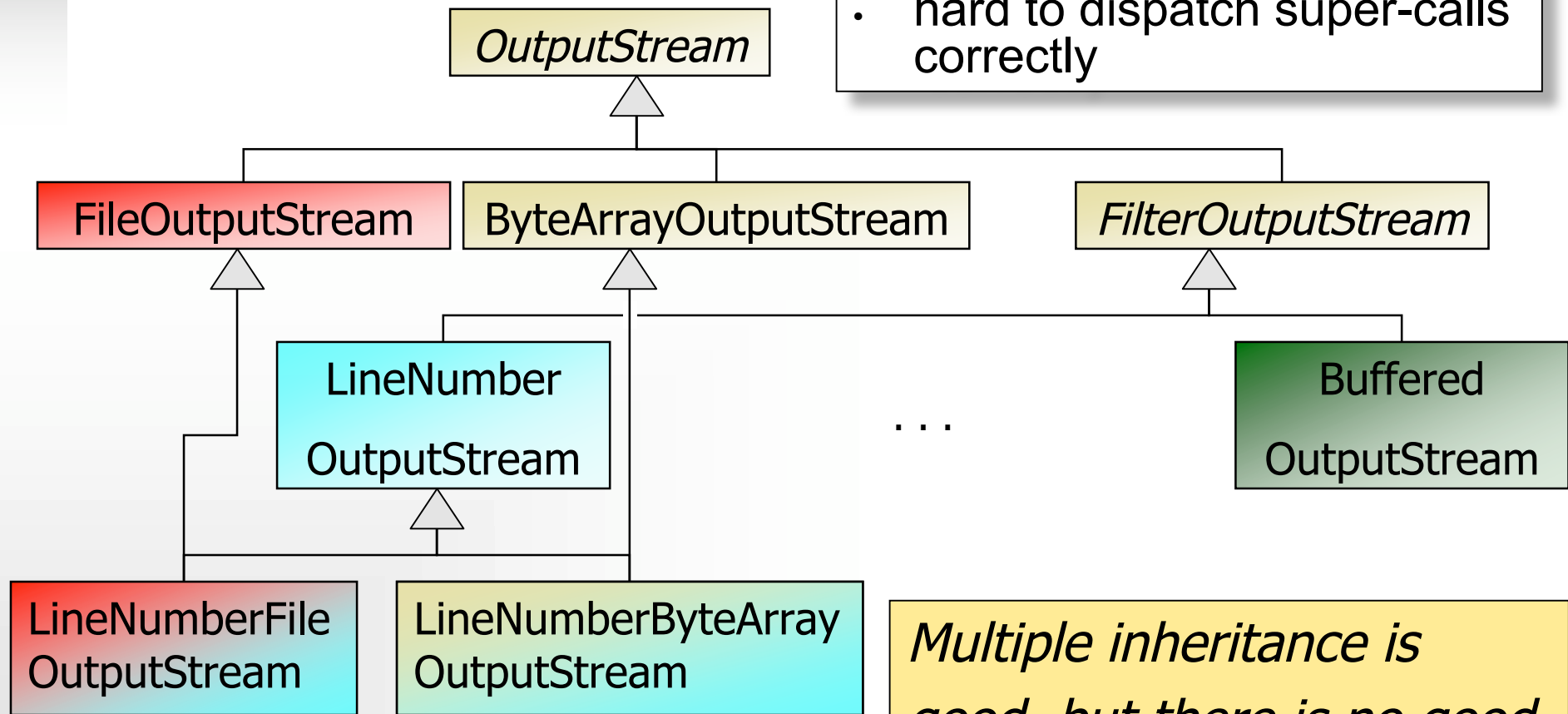
- static
- non-reusable extensions
- code duplication:
maintenance nightmare
- exponential growth of no. of classes



Inheritance Disadvantages

Multiple Inheritance

- static
- naming conflicts
- hard to dispatch super-calls correctly



Multiple inheritance is good, but there is no good way to do it ... A. Snyder

Decorator Advantages

Avoidance of Incoherent Classes

- incoherence: feature-laden classes high up in the hierarchy
 - this also breaks encapsulation
- pay-as-you-go approach
 - don't bloat, but extend using fine-grained **Decorator** classes
- functionality can be composed from simple pieces
 - thus, an application does not need to pay for features it doesn't use
- a fine-grained **Decorator** hierarchy is easy to extend

Decorator Disadvantages

Lots of Little Objects

- a design that uses Decorator often results in systems composed of lots of little objects that all look alike
- objects differ only in the way they are interconnected, not in their class or in the value of their variables
- these systems are easy to customise by those who understand them, they can be hard to learn and debug

Object Identity

- a decorator and its component aren't identical
 - from an object identity point of view, a decorated component is not identical to the component itself
- shouldn't rely on object identity when you use decorators

Decorator Disadvantages

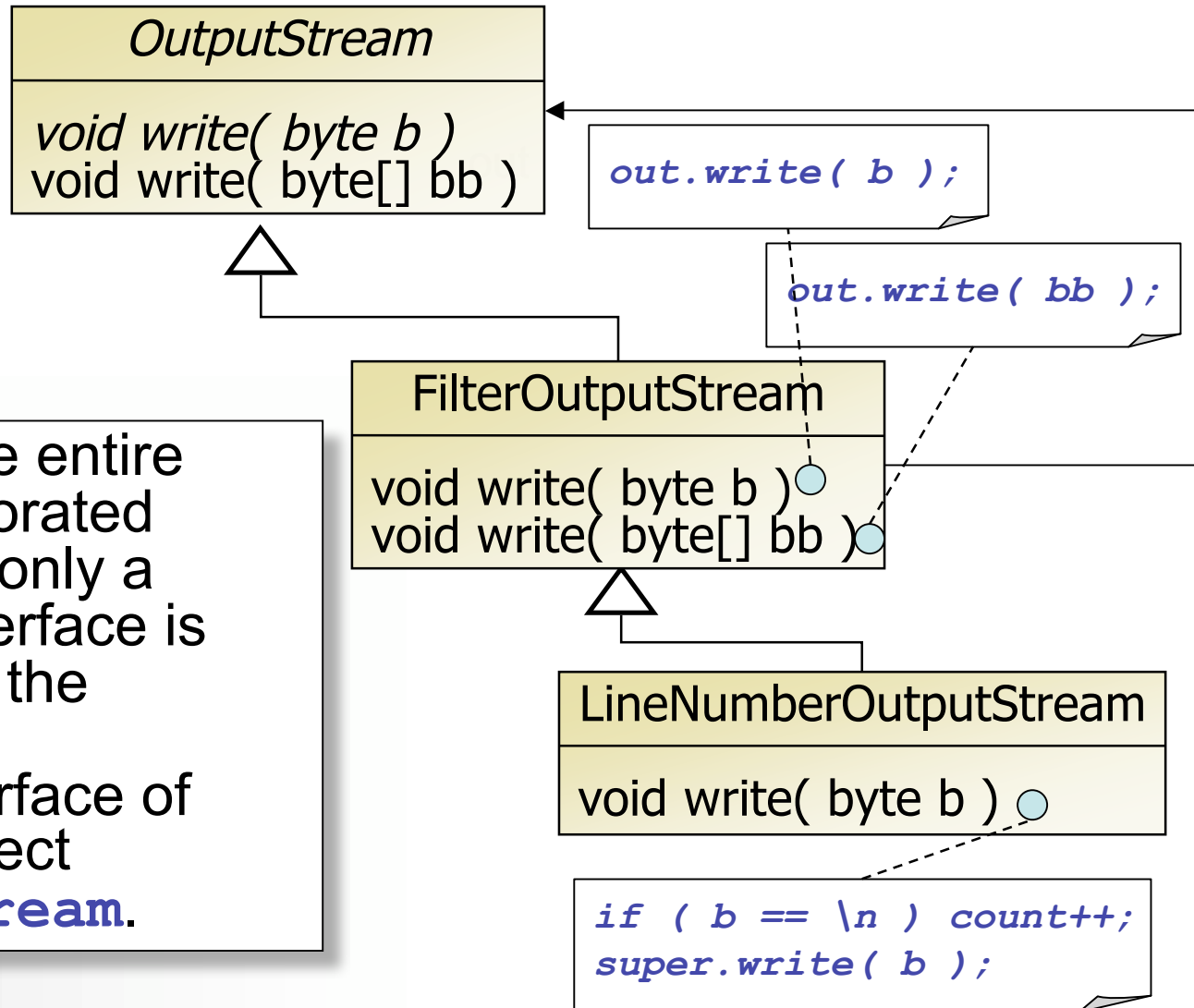
Object Identity

Example: Decorator in java.io Package

- a stream is normally addressed via the outermost Decorator
- sometimes, a reference to one of the internal objects is maintained and operated on
- this operation shouldn't include actual reads or writes
 - as a matter of good style, all `read()` operations are performed only to the head decorator in the composite stream object
- *reading from an internal object breaks the uncomplicated illusion of a single object accessed via a single reference, and makes the code more difficult to understand*

Decorator Disadvantages

Fragile Base Class Problem



Need to redefine the entire interface of the decorated abstraction, even if only a small part of the interface is actually affected by the decoration.
Changes in the interface of **OutputStream** affect **FilterOutputStream**.

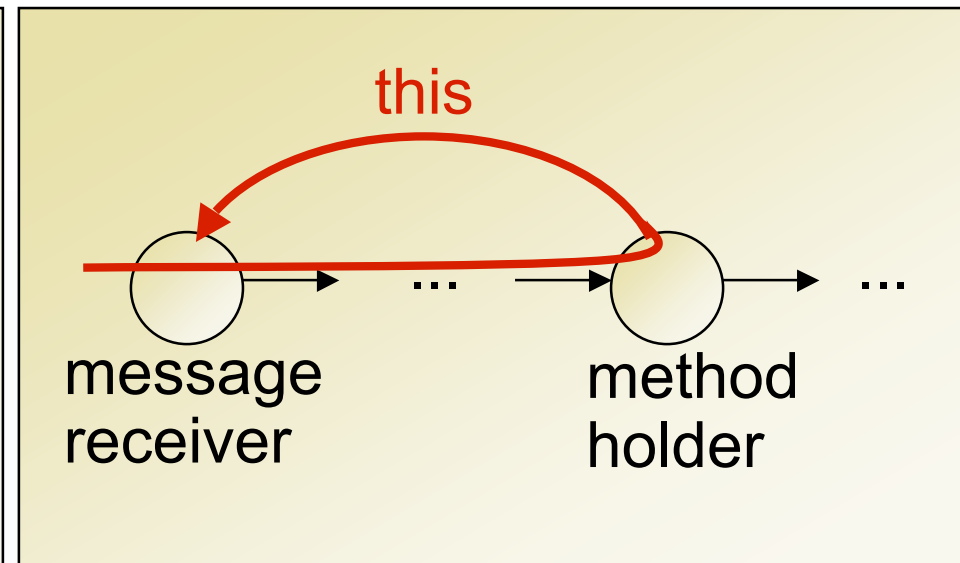
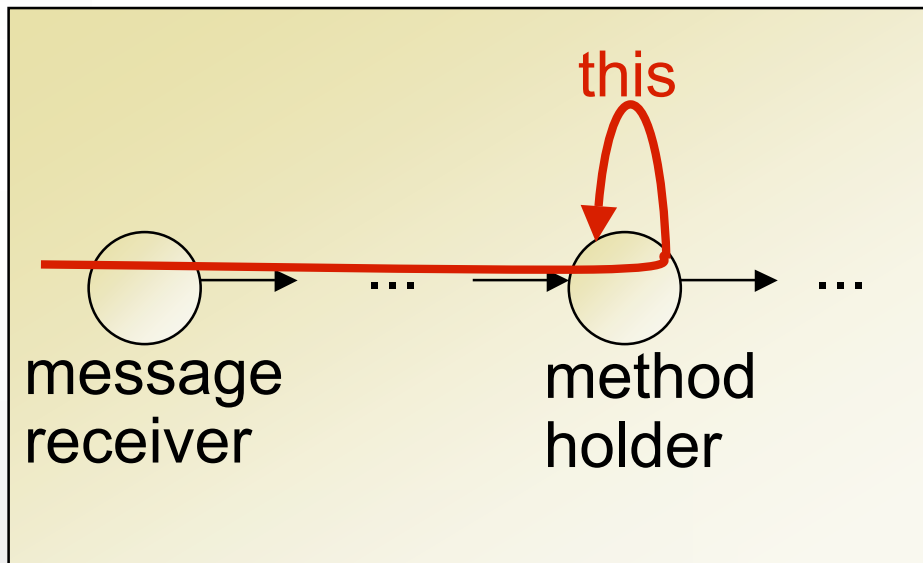
Decorator Disadvantages

No Late-Binding of **this**

delegation versus forward semantics

Forwarding

Delegation



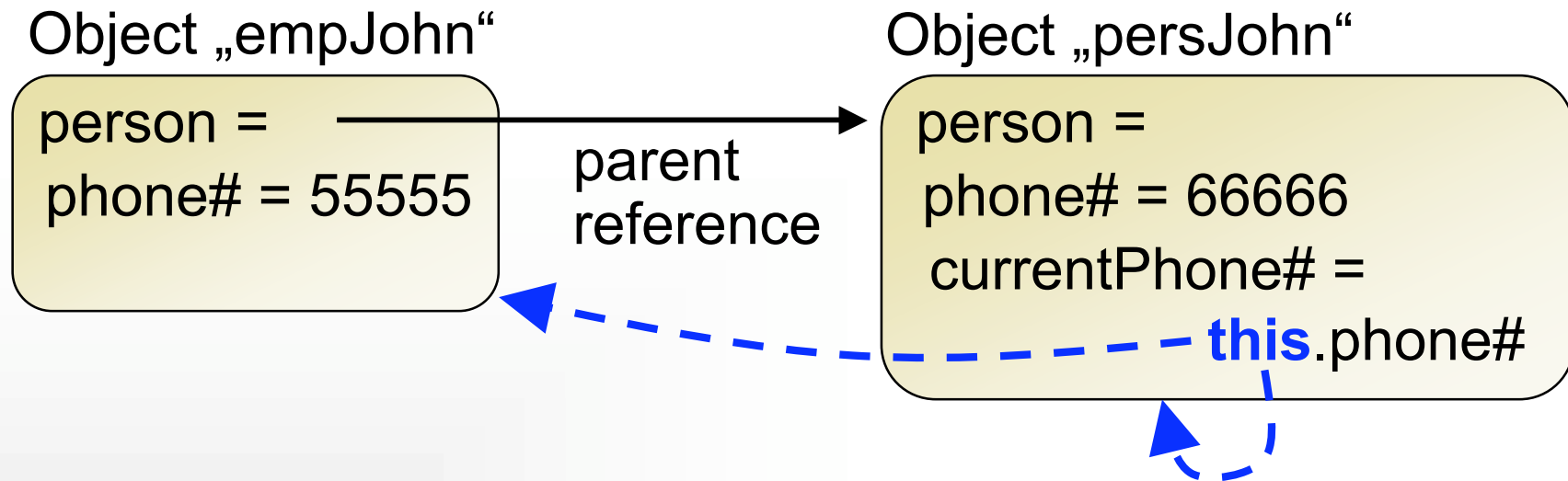
automatic forwarding with binding of **this** to method holder: ask an object to do something *on its own*

automatic forwarding with binding of **this** to message receiver: ask an object to do something *on behalf of the message receiver*

Decorator Disadvantages

No Late-Binding of **this**

delegation versus forward semantics

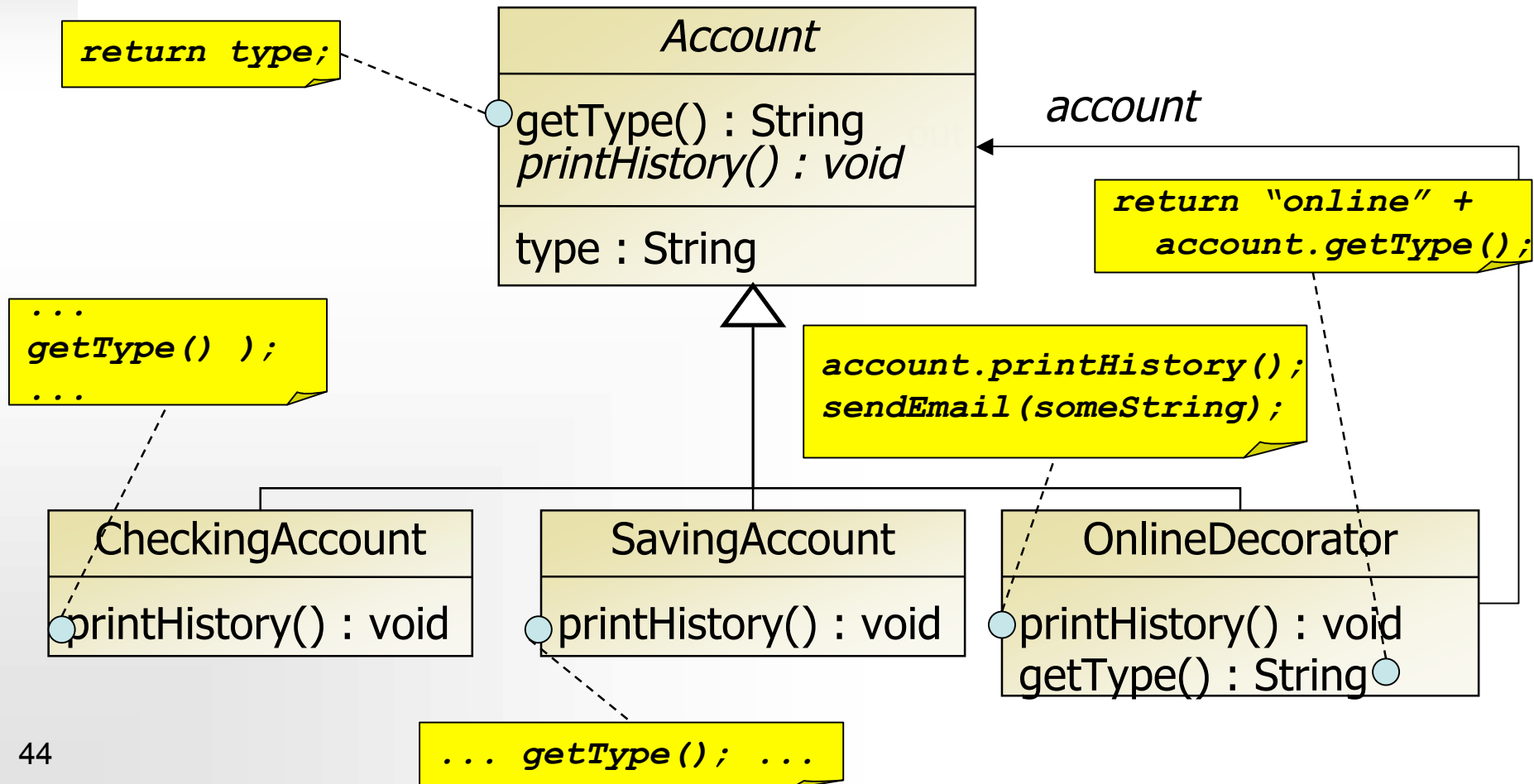


Forwarding:	empJohn.currentPhone# -> 66666
Delegation:	empJohn.currentPhone# -> 55555

Decorator Disadvantages

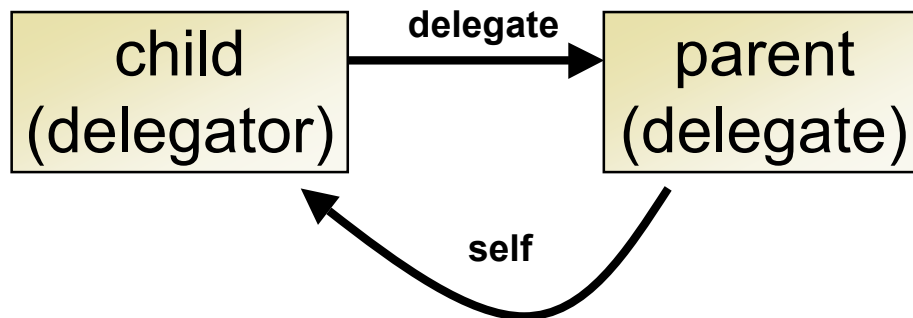
No Late-Binding of **this**

Java only supports forward semantics!



Simulating Delegation

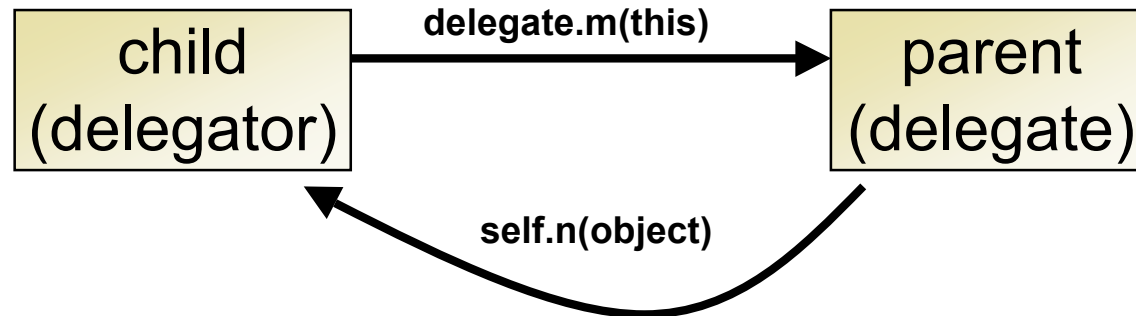
Storing References to Delegates



- **"self"** reference stored in delegate object
 - points back to delegating object
- only one delegator per delegate possible
 - e.g., sharing of **System.out** wouldn't work
- only if delegator is initial receiver
 - delegates cannot be used as receivers of their own messages

Simulating Delegation

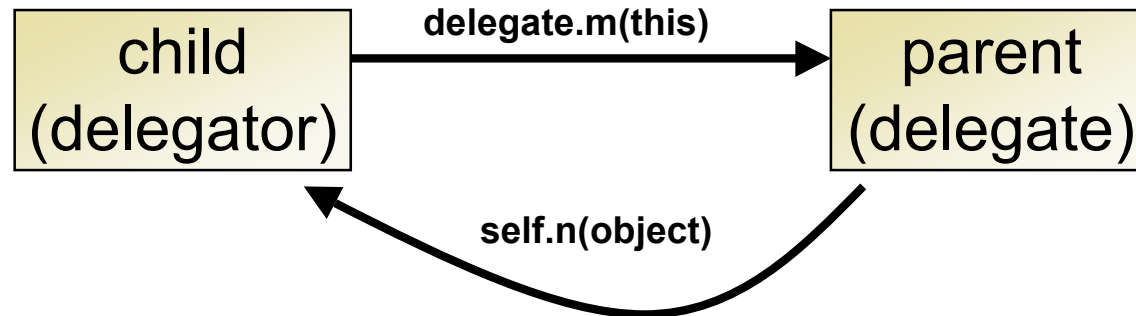
Passing **self** as Method Argument



- normal message passing: `obj.msg(args)` is replaced by `obj.msg(this, args)`
- messages to `this` are redirected to first argument: `this.msg(args)` is replaced by `self.msg(self, args)`

Simulating Delegation

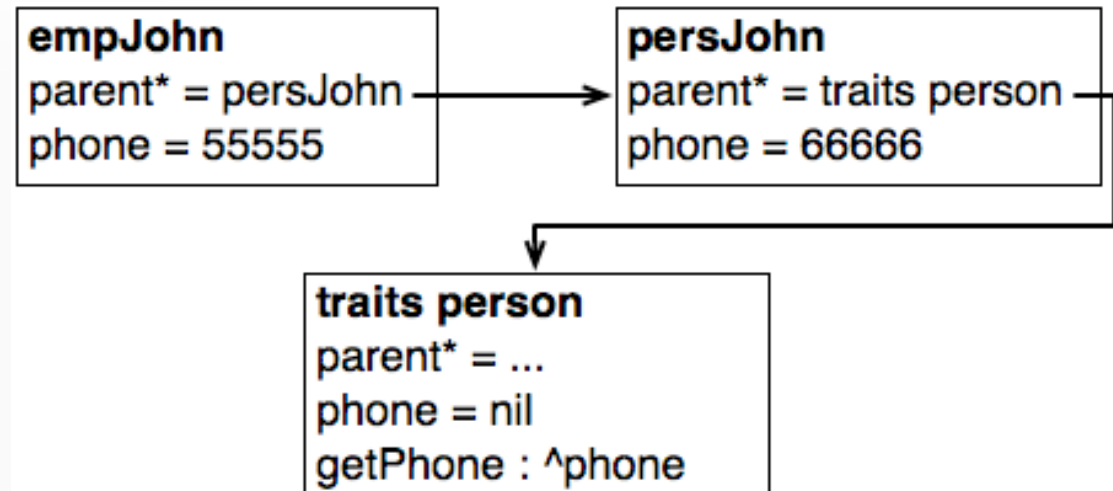
Passing **self** as Method Argument



- works for sharing as well as recursive delegation
- requires extensive modifications
- awareness of decorator, difficult to do transparently
- main problem: typing
 - what is the type of **self**?

Language Support for Delegation

- prominent representatives: **Self** and **Lava**
- **Self**: Smalltalk derivative
 - purely object-oriented: **no classes**
 - prototype-based OO
 - delegation as only concept to facilitate late binding
- no classes?
 - instance creation by prototype cloning
 - **traits objects** define shared behaviour and structure



Language Support for Delegation

- **Lava**: extension to Java with delegation support
 - define **forwarding and delegation** semantics for object compositions
 - forwarding is called **consultation** in Lava
 - type-safe

```
class OnlineDecorator extends Account {
    private mandatory delegatee Account account;
    public OnlineDecorator(Account a) {
        account = a;
    }
    public getType() {
        return "online " + account.getType();
    }
    public printHistory() {
        account.printHistory();
        sendEmail(...);
    }
}
```

Decorator: Implementation Issues

Keeping Component Lightweight

- keep the common **Component** class lightweight: it should focus on defining an interface, not on storing data
- defining the data representation should be deferred to subclasses; otherwise the complexity of the **Component** class might make the decorators too heavyweight to use in quantity
- putting a lot of functionality into **Component** also increases the probability that concrete subclasses will pay for features they don't need
- these issues require **pre-planning!**
 - difficult to apply decorator pattern to 3rd-party component class

Decorator: Implementation Issues

- **interface conformance**: a decorator's interface must conform to the interface of the component it decorates
 - **ConcreteDecorator** classes must therefore inherit from a common class (C++) or implement a common interface (Java)
- **no need** to define an abstract **Decorator** class when you only need to add one responsibility
 - that's often the case when you're dealing with an existing class hierarchy rather than designing a new one
 - can merge Decorator's responsibility for forwarding requests to the component into the **ConcreteDecorator**