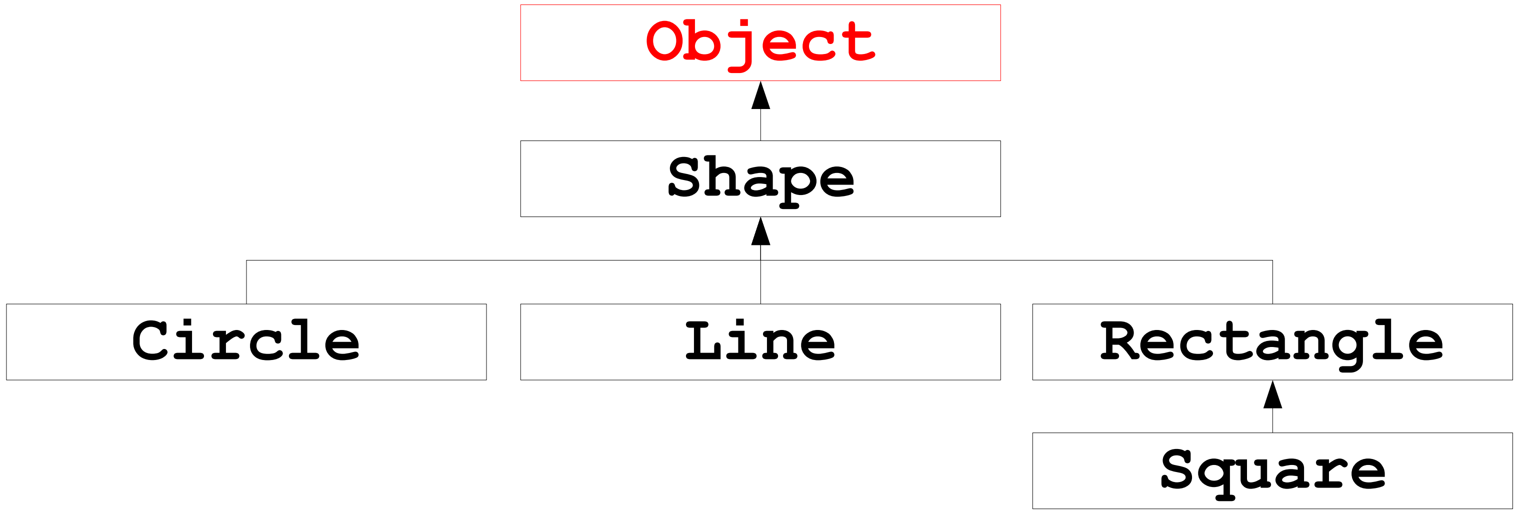
Polymorphism

- Polymorphism
  - Goals
  - Upcast and downcast revisited
- Static and dynamic type
- Method binding
  - Static binding
  - Dynamic binding
- Example using Polymorphism
  - A polymorphic field (the *state design pattern*)
- Abstract classes
  - The *composite design pattern* revisited
    - Using polymorphism
    - Using abstract classes

Class Hierarchies in Java, Revisited

- Class **`Object`** is the root of the inheritance hierarchy in Java.
- If no superclass is specified a class inherits *implicitly* from **`Object`**.
- If a superclass is specified *explicitly* the subclass will inherit indirectly from **`Object`**.



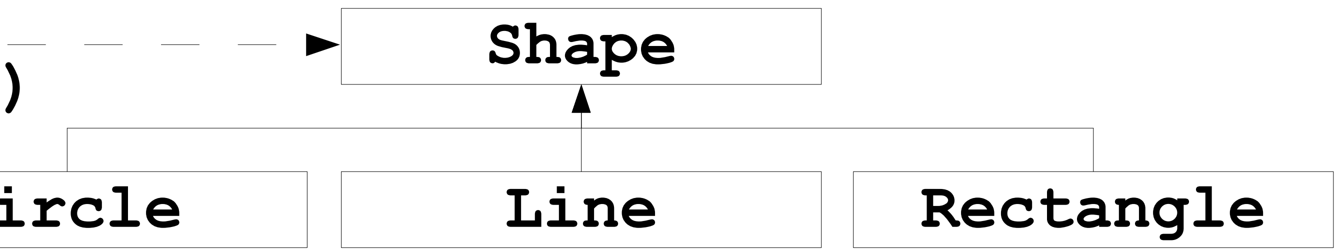
Polymorphism, Informal

- In a bar you say "I want a beer!"
  - What ever beer you get is okay because your request was very generic
- In a bar you say "I want a Samuel Adams Cherry Flavored beer!"
  - If you do not exactly get this type of beer you are allowed to complain

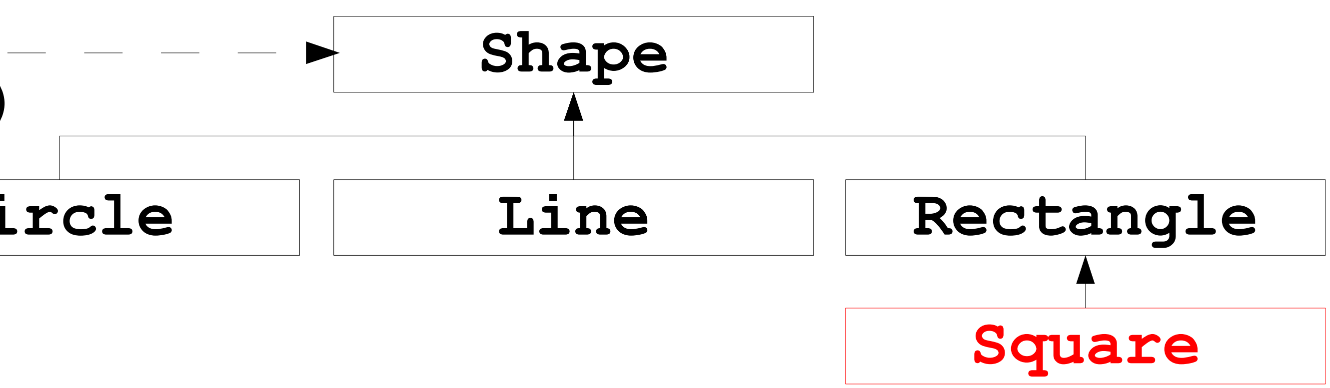
- In chemistry they talk about polymorph materials as an example H_20 is polymorph (ice, water, and steam).

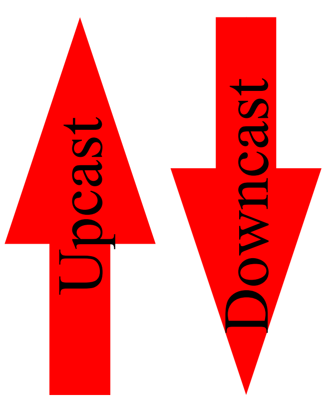
Goals Polymorphism

```
// abstraction (and substitutability)
Shape s;
s.draw()
s.resize()
```




```
// extensibility
Shape s;
s.draw()
s.resize()
```





Advantages/Disadvantages of Upcast/Downcast

- Advantages
  - Code is simpler to write (and read)
  - Uniform interface for clients, i.e., type specific details only in class code, not in the client code
  - Change in types in the class does not effect the clients
    - If type change within the inheritance hierarchy
  - Used extensively in object-oriented programs
    - Many upcast to **`Object`** in the standard JDK library

- Disadvantages
  - Must explicitly *downcast* if type details needed in client after object has been handled by the standard library (very annoying sometimes).

```
Shape s = new Line(); // implicit upcast
Line l = (Line) s;    // explicit downcast
```

Polymorphism

- *Polymorphism:* "The ability of a variable or argument to refer at run-time to instances of various classes" [Meyer pp. 224].

```
Shape s = new Shape();
Circle c = new Circle();
Line l = new Line();
Rectangle r = new Rectangle();

s = l;          // is this legal?
l = s;          // is this legal?
l  = (Line)s    // is this legal?
```

- The assignment **`s = l`** is legal if the static type of **`l`** is **`Shape`** or a subclass of **`Shape`**.
- This is *static type checking* where the type comparison rules can be done at compile-time.
- Polymorphism is constrained by the inheritance hierarchy.

Static and Dynamic Type

- The *static type* of a variable/argument is the declaration type.
- The *dynamic type* of a variable/argument is the type of the object the variable/argument refers to.

```
class A{
  // body
}
class B extends A{
  // body
}

public static void main(String args[]){
   A x;              // x static type A
   B y;              // y static type B

   x = new A();      // x dynamic type A
   y = new B();      // y dynamic type B
   x = y;            // x dynamic type B
}
```

Method Binding

```
class A {
   void doSomething(){
      ...
   }
}
```

```
class B extends A {
   void doSomething (){
      ...
   }
}
```

```
A x = new A();
B y = new B();
x = y;
x.doSomething(); // on class A or class B?
```

- *Binding*: Connecting a method call to a method body.
- *Dynamic binding*: The dynamic type of **x** determines which method is called (also called *late binding*).
  - Dynamic binding is not possible without polymorphism.
- *Static binding*: The static type of **x** determines which method is called (also called *early binding*).

Dynamic Binding, Example

```
public class Shape {
   void draw() { System.out.println ("Shape"); }
}
public class Circle extends Shape {
   void draw() { System.out.println ("Circle"); }
}
public class Line extends Shape {
   void draw() { System.out.println ("Line"); }
}
public class Rectangle extends Shape {
   void draw() {System.out.println ("Rectangle"); }
}

public static void main(String args[]){
   Shape[] s = new Shape[3];
   s[0] = new Circle();
   s[1] = new Line();
   s[2] = new Rectangle();
   for (int i = 0; i < s.length; i++){
      s[i].draw(); // prints Circle, Line, Rectangle
   }
}
```

Dynamic Binding and Constructors

```
public class A { // example from inheritance lecture
    public A(){
        System.out.println("A()");
        // when called from B the B.doStuff() is called
        doStuff();
    }
    public void doStuff(){ System.out.println("A.doStuff()"); }
}

public class B extends A{
    int i = 7;
    public B(){ System.out.println("B()"); }
    public void doStuff(){System.out.println("B.doStuff() " + i);}
}

public class Base{
    public static void main(String[] args){
        B b = new B();
        b.doStuff();
    }
}
```

```
//prints
A()
B.doStuff() 0
B()
B.doStuff() 7
```

Dynamic Binding and **`private`** Methods

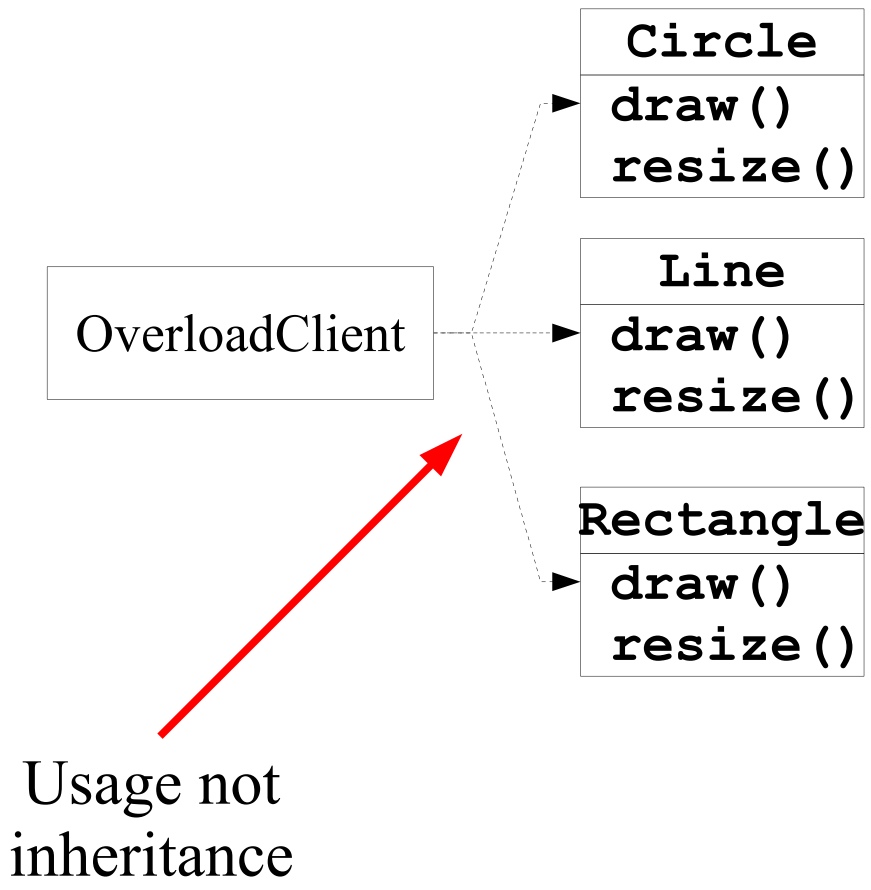
```
class Shape {
   void draw() { System.out.println ("Shape"); }
   private void doStuff() {
      System.out.println("Shape.doStuff()");
   }
}
class Rectangle extends Shape {
   void draw() { System.out.println ("Rectangle"); }
   public void doStuff() {
      System.out.println("Rectangle.doStuff()");
    }
}

public class PolymorphShape {
   public static void polymorphismPrivate(){
      Rectangle r = new Rectangle();
      r.doStuff();   // okay part of Rectangle interface
      Shape s = r;   // upcast
      s.doStuff();   // not allowed, compiler error
   }
}
```

Why Polymorphism and Dynamic Binding?

- Separate interface from implementation.
  - Encapsulation
  - What we are trying to achieve in object-oriented programming!
- Allows programmers to isolate type specific details from the main part of the code.
  - Client programs only use the method provided by the **`Shape`** class in the shape hierarchy example.
- Code is simpler to write and to read.
  - Abstraction, abstraction, and abstraction!
- Can change types (and add new types) without the changes propagates to existing code.

Overloading vs. Polymorphism (1)

- Has not yet discovered that the **`Circle`**, **`Line`**, and **`Rectangle`** classes are related. (not very realistic but just to show the idea).

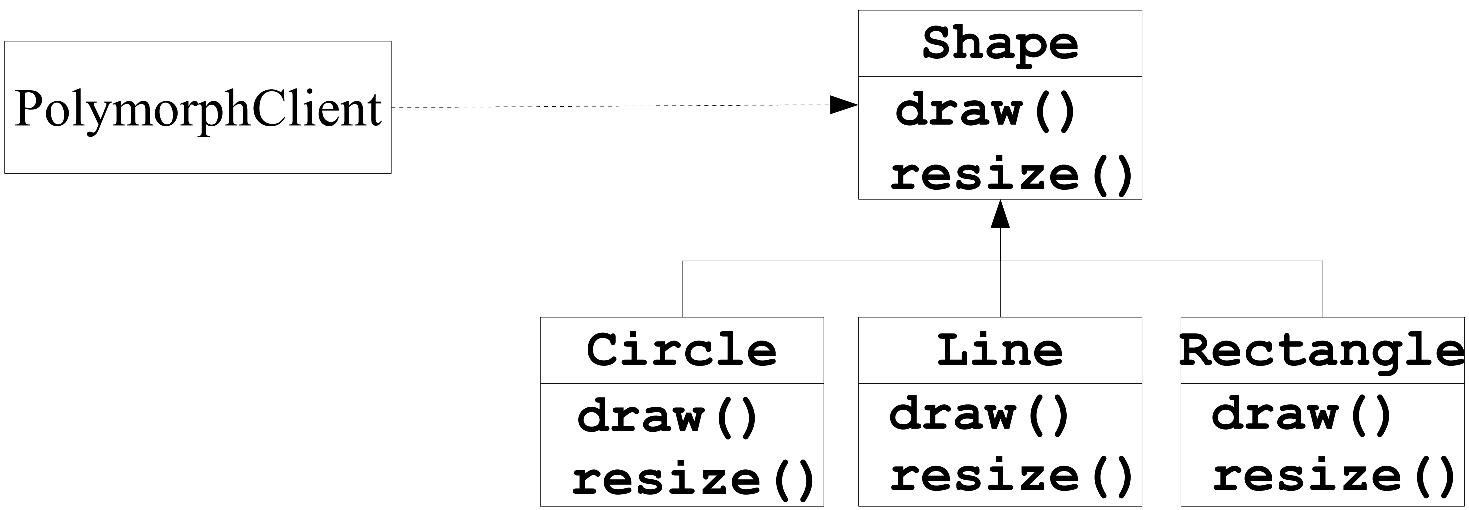


Overloading vs. Polymorphism (2)

```
class Circle {
    void draw() { System.out.println("Circle"); }}
class Line {
    void draw() { System.out.println("Line"); }}
class Rectangle {
    void draw() { System.out.println("Rectangle"); }}

public class OverloadClient{
    // make a flexible interface by overload and hard work
    public void doStuff(Circle c){ c.draw(); }
    public void doStuff(Line l){ l.draw(); }
    public void doStuff(Rectangle r){ r.draw(); }

    public static void main(String[] args){
        OverloadClient oc = new OverloadClient();
        Circle ci = new Circle();
        Line li = new Line();
        Rectangle re = new Rectangle();
        // nice encapsulation from client
        oc.doStuff(ci); oc.doStuff(li); oc.doStuff(re);
    }
}
```

Overloading vs. Polymorphism (3)

- Discovered that the **`Circle`**, **`Line`**, and **`Rectangle`** class are related are related via the general concept **`Shape`**
- Client only needs access to base class methods.

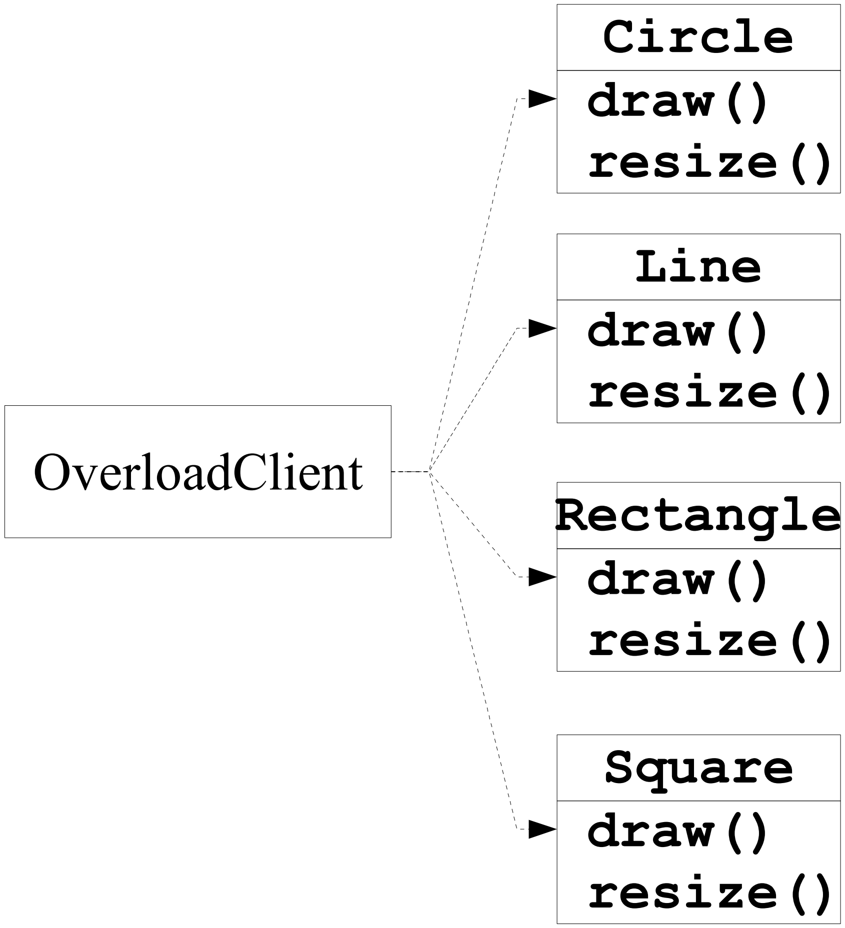


Overloading vs. Polymorphism (4)

```
class Shape {
   void draw() { System.out.println("Shape"); }}
class Circle extends Shape {
    void draw() { System.out.println("Circle"); }}
class Line extends Shape {
    void draw() { System.out.println("Line"); }}
class Rectangle extends Shape {
    void draw() { System.out.println("Rectangle"); }}

public class PolymorphClient{
    // make a really flexible interface by using polymorphism
    public void doStuff(Shape s){ s.draw(); }

    public static void main(String[] args){
        PolymorphClient pc = new PolymorphClient();
        Circle ci = new Circle();
        Line li = new Line();
        Rectangle re = new Rectangle();
        // still nice encapsulation from client
        pc.doStuff(ci); pc.doStuff(li); pc.doStuff(re);
    }
}
```

Overloading vs. Polymorphism (5)

- Must extend with a new class **`Square`** and the client has still not discovered that the **`Circle`**, **`Line`**, **`Rectangle`**, and **`Square`** classes are related.

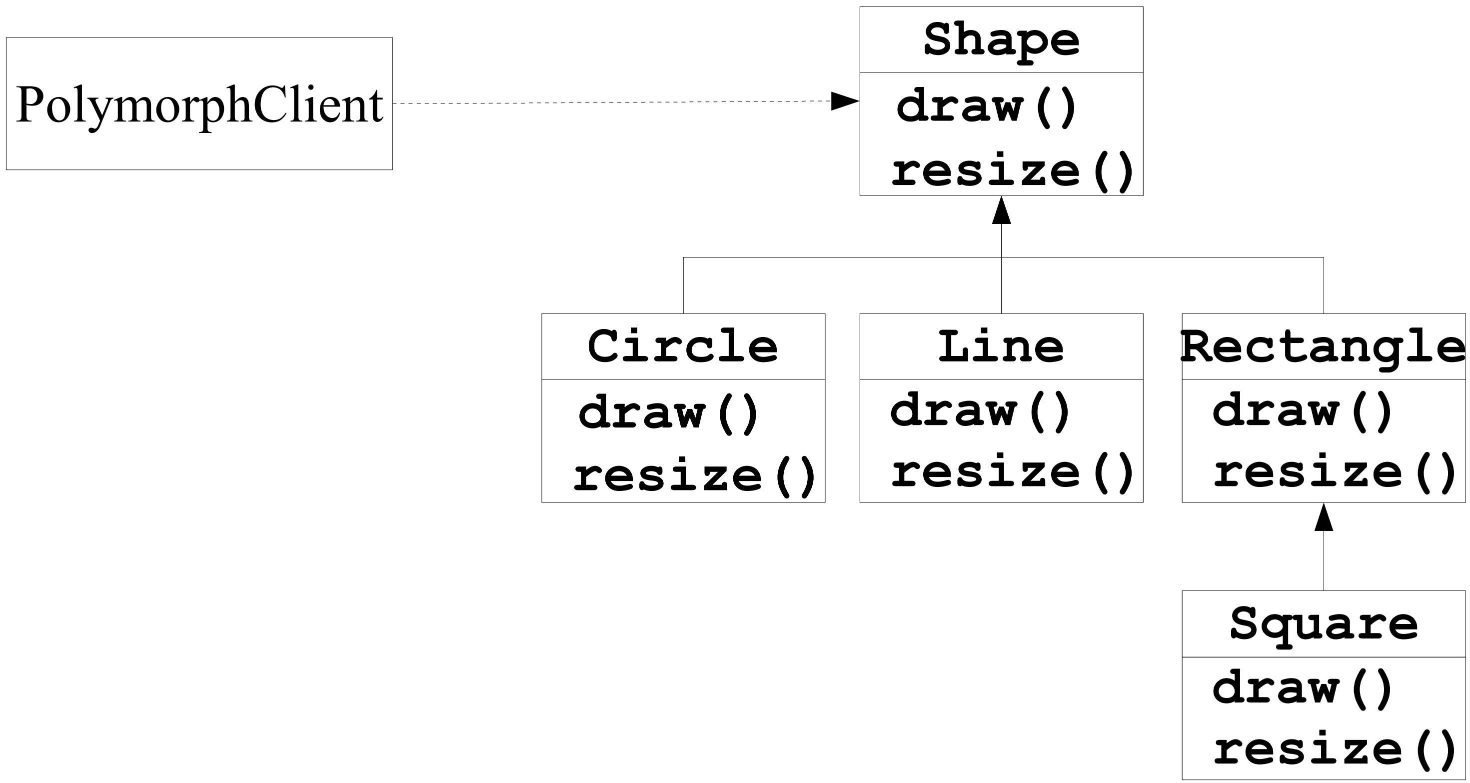


Overloading vs. Polymorphism (6)

```
class Circle {
    void draw() { System.out.println("Circle"); }}
class Line {
    void draw() { System.out.println("Line"); }}
class Rectangle {
    void draw() { System.out.println("Rectangle"); }}
class Square {
    void draw() { System.out.println("Square"); }}

public class OverloadClient{
    // make a flexible interface by overload and hard work
    public void doStuff(Circle c){ c.draw(); }
    public void doStuff(Line l){ l.draw(); }
    public void doStuff(Rectangle r){ r.draw(); }
    public void doStuff(Square s){ s.draw(); }

    public static void main(String[] args){
        <snip>
        // nice encapsulation from client
        oc.doStuff(ci); oc.doStuff(li); oc.doStuff(re);
    }
}
```

Overloading vs. Polymorphism (7)

- Must extend with a new class **`Square`** that is a subclass to **`Rectangle`**.



Overloading vs. Polymorphism (8)

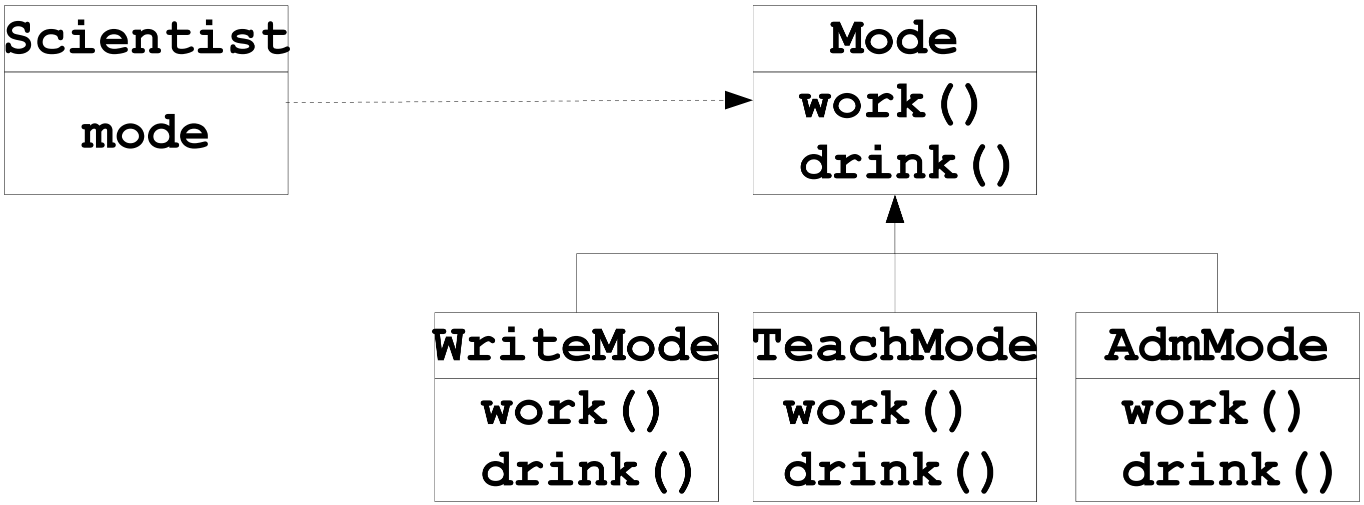
```
class Shape {
   void draw() { System.out.println("Shape"); }}
class Circle extends Shape {
    void draw() { System.out.println("Circle"); }}
class Line extends Shape {
    void draw() { System.out.println("Line"); }}
class Rectangle extends Shape {
    void draw() { System.out.println("Rectangle"); }}
class Square extends Rectangle {
    void draw() { System.out.println("Square"); }}

public class PolymorphClient{
    // make a really flexible interface by using polymorphism
    public void doStuff(Shape s){ s.draw(); }

    public static void main (String[] args){
        <snip>
        // still nice encapsulation from client
        pc.doStuff(ci); pc.doStuff(li); pc.doStuff(re);
   }
}
```

The Opened/Closed Principle

- Open
  - The class hierarchy can be extended with new specialized classes
- Closed
  - The new classes added do not effect old clients
  - The superclass interface of the new classes can be used by old clients
- This is made possible via
  - Polymorphism
  - Dynamic binding
    - Try to do this in C or Pascals!

A Polymorph Field

- A scientist does three very different things (modes)
  - Writes paper (and drinking coffee)
  - Teaches classes (and drinking water)
  - Administration (and drinking tea)
- The implementation of each is assumed to be very complex
- Must be able to change dynamically between these modes



Implementing a Polymorph Field

```
public class Mode{
   public void work(){ System.out.println("");}
   public void drink(){ System.out.println("");}
}

public class WriteMode extends Mode{
   public void work(){ System.out.println("write");}
   public void drink(){ System.out.println("coffee");}
}

public class TeachMode extends Mode{
   public void work(){ System.out.println("teach");}
   public void drink(){ System.out.println("water");}
}

public class AdmMode extends Mode{
   public void work(){ System.out.println("administrate");}
   public void drink(){ System.out.println("tea");}
}
```

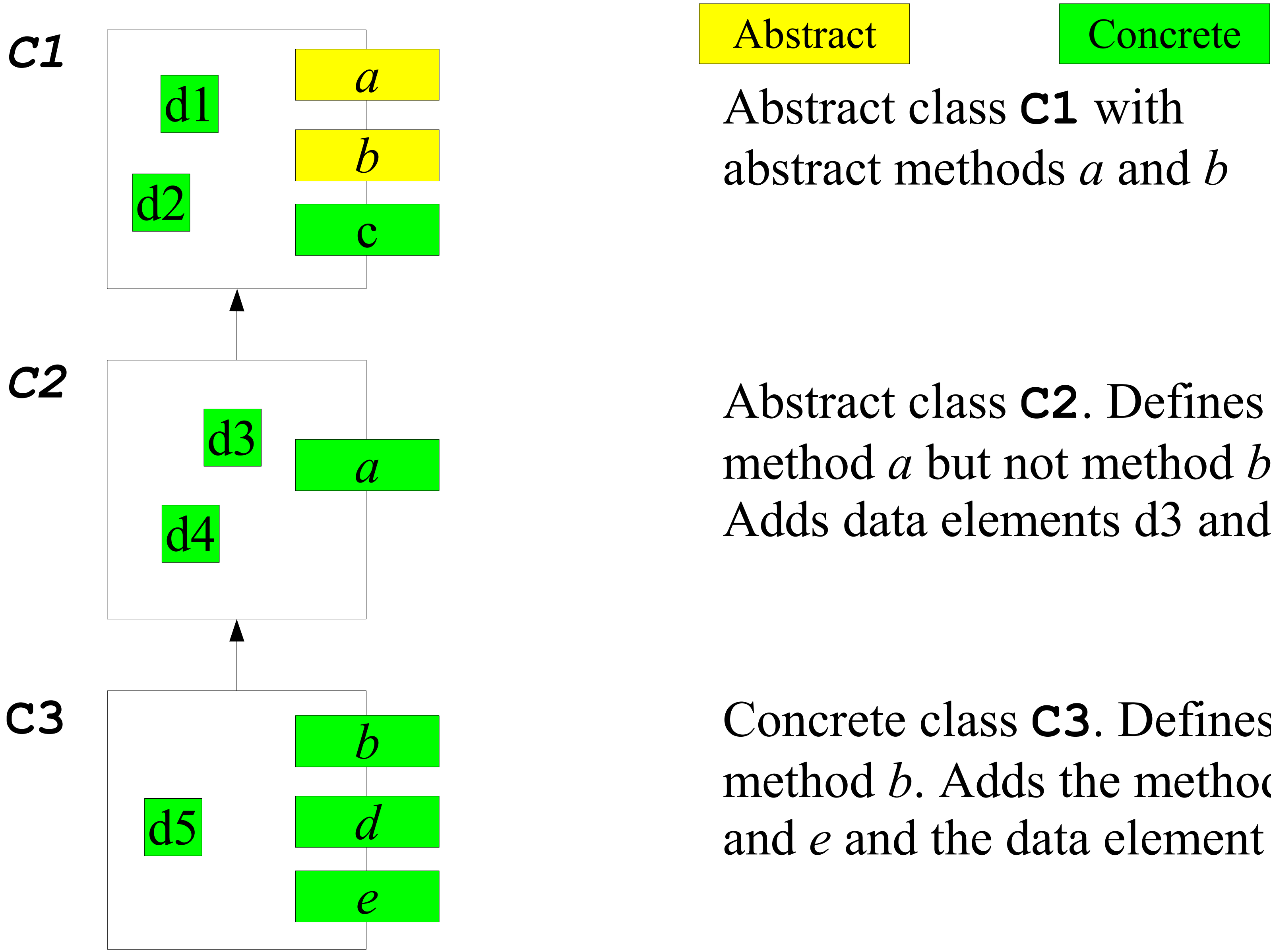
Implementing a Polymorph Field, cont.

```
public class Scientist{
   private Mode mode;
   public Scientist(){
      mode = new WriteMode(); /* default mode */
   }
   // what scientist does
   public void doing() { mode.work();}
   public void drink() { mode.drink();}

   // change modes methods
   public void setWrite() { mode = new WriteMode();}
   public void setTeach() { mode = new TeachMode();}
   public void setAdministrate() { mode = new AdmMode();}

   public static void main(String[] args){
      Scientist einstein = new Scientist();
      einstein.doing();
      einstein.setTeach();
      einstein.doing();
   }
}
```

Evaluation of the Polymorph Field

- Can change modes dynamically
  - Main purpose!
- Different modes are isolated in separate classes
  - Complexity is reduced (nice side-effect)
- Clients of the **`Scientist`** class can see the **`Mode`** class (and its subclasses).
  - This may unnecessarily confuse these clients.
- **`Scientist`** class *cannot* change mode added after it has been compiled, e.g., **`SleepMode`**.
- Can make instances of **`Mode`** class. This should be prevented.
  - We will do this next!
- The *state design pattern*
  - Nice design!

Abstract Class and Method, Example



Abstract class **`C1`** with abstract methods *a* and *b*

Abstract class **`C2`**. Defines method *a* but not method *b*. Adds data elements d3 and d4

Concrete class **`C3`**. Defines method *b*. Adds the methods *d* and *e* and the data element d5.

Abstract Classes and Methods in Java, Example

```
public abstract class Mode{
   // abstract methods
   public abstract void work(); // no body
   public abstract void drink();

    // concrete methods
    public String toString(){
      return "Mode";
  }
}
```

- An abstract method has no method body.
- It is *not* possible to make instances of abstract classes.
- Abstract method are defined in subclasses of the abstract class.
  - No changes in **Mode**'s subclasses

Abstract Classes in Java

```
public abstract class Mode {
    // class body
    <snip>
}
```

- Classes with abstract methods *must* declared abstract.
- Classes without abstract methods *can* be declared abstract.
- A subclass to a concrete superclass can be abstract.
- Constructors can be defined on abstract classes.
- Instances of abstract classes cannot be made.

- Abstract fields not possible.

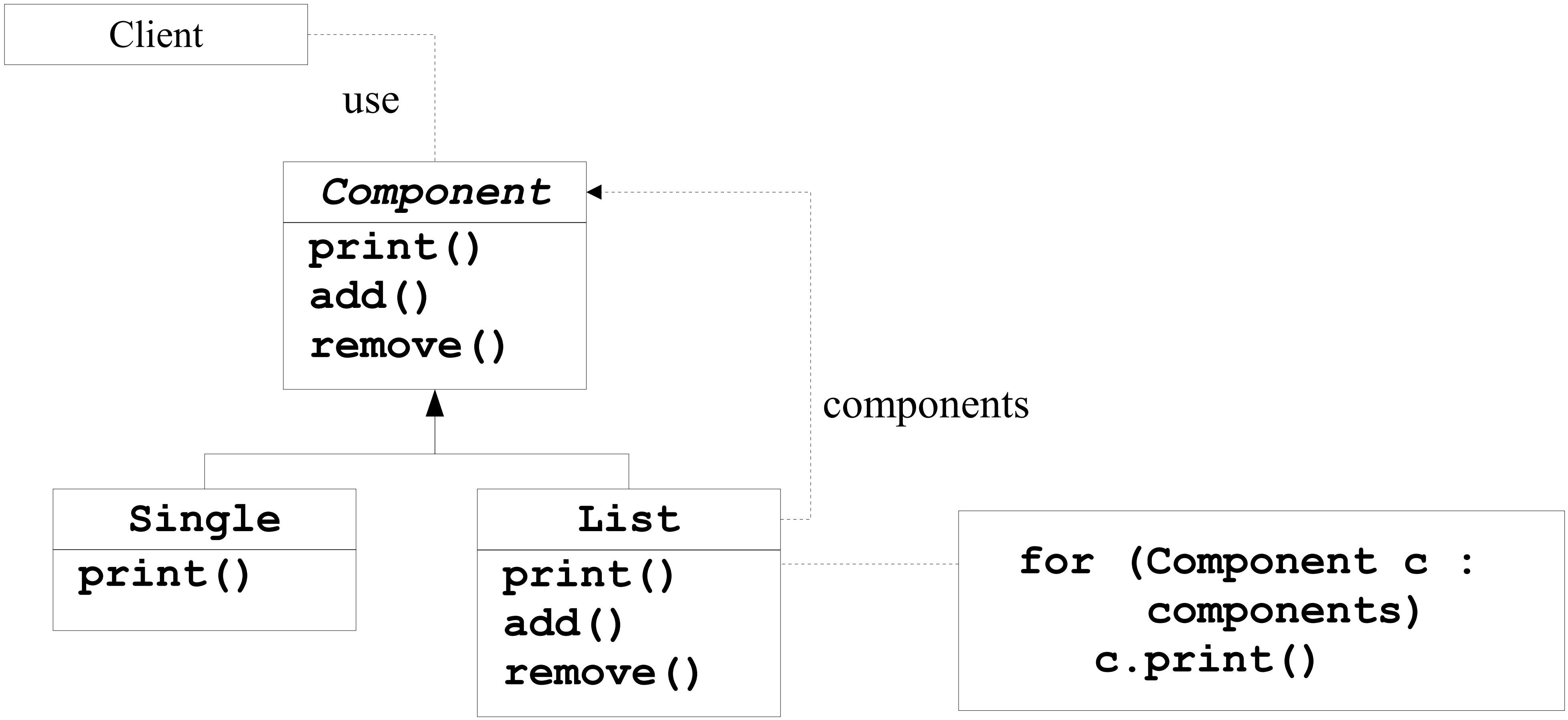
Abstract Methods in Java

```
public abstract class Mode{
    public abstract void work();
    public abstract void drink();
    <snip>
}
```

- A method body does not have be defined.
- Abstract method are overwritten in subclasses.
- Idea taken directly from C++
  - pure virtual function

- You are saying: “The object should have these properties I just do not know how to implement these properties at this level of abstraction.”

The Composite Design Pattern



- ***Component*** class in *italic* means abstract class
- **Single** typically called *leaf*
- **List** typically called *composite*

Implementation of The Composite Pattern

```
public abstract class Component{
   public abstract void print(); // no body
   public void add(Component c){ // still concrete!
      System.out.println("Do not call add on me!");}
   public void remove(Component c){ // still concrete!
      System.out.println("Do not call remove on me!");}
}

public class Single extends Component{
   private String name;
   public Single(String n){ name = n;}
   public void print(){ System.out.println(name); }
}

public class List extends Component{
   private Component[] comp; private int count;
   public List(){ comp = new Component[100]; count = 0; }
   public void print(){ for(int i = 0; i <= count - 1; i++){
         comp[i].print(); // polymorphism
      }
   }
   public void add(Component c){ comp[count++] = c;}
}
```

Evaluation of the Composite Design Pattern

- Made **`List`** and **`Single`** classes look alike when printing from the client's point of view.
  - The main objective!
- Can make instances of **`Component`** class, not the intension
  - Can call dummy add/remove methods on these instances (FIXED)
- Can call add/remove method of **`Single`** objects, not the intension. (CANNOT BE FIXED).
- Fixed length, not the intension.
- Nice design!

Summary

- Polymorphism an object-oriented “switch” statement.
- Polymorphism should strongly be preferred over overloading
  - Must simpler for the class programmer
  - Identical (almost) to the client programmer
- Polymorphism is a prerequisite for dynamic binding and central to the object-oriented programming paradigm.
  - Sometimes polymorphism and dynamic binding are described as the same concept (this is inaccurate).
- Abstract classes
  - Complete abstract class no methods are abstract but instantiation does not make sense.
  - Incomplete abstract class, some method are abstract.

Abstract Classes and Methods in Java, Example

```
// [Source: Kurt Nørmark]
public abstract class Mode{
  // abstract methods
  public abstract push(Object el);
  public abstract void pop(); // note no return value
  public abstract Object top();
  public abstract boolean full();
  public abstract boolean empty();
  public abstract int size();

  // concrete methods
  public void toggleTop(){
    if (size() >= 2){
      Object topEl1 = top();  pop();
      Object topEl2 = top();  pop();
      push(topEl1); push(topEl2);
    }
  }
  public String toString(){
    return "Stack";
  }
}
```

Abstract Methods in Java, Example

```
public abstract class Number {
    public abstract int intValue();
    public abstract long longValue();
    public abstract double doubleValue();
    public abstract float floatValue();
    public byte byteValue(){
        // method body
    }
    public short shortValue(){
        // method body
    }
}
```