Comp 401
Inheritance vs. Delegation

Indiscriminate Use Of Inheritance Considered Harmful

Instructor: Prasun Dewan

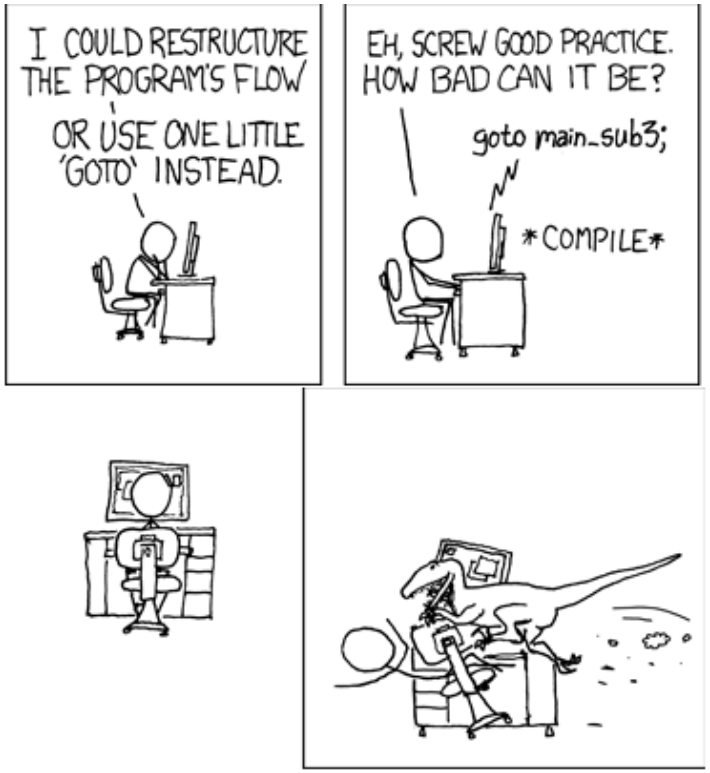
Prerequisite

- Inheritance: Abstract Classes
- Inheritance: Virtual Abstract Factory Methods
- Model View Controller

Computer Science

- First we teach you something.
- Then we say it has bad consequences.

Go To Statement



GO TO <STATEMENT ID>

GOTOs Required cleverness and hence were cool

Go To Statement Considered Harmful
Edsger W. Dijkstra

Reprinted from *Communications of the ACM*, Vol. 11, No. 3, March 1968, pp. 147-148. Copyright © 1968

This is a digitized copy derived from an ACM copyrighted work. It is not guaranteed to be an accurate copy of

Pascal: Go To supported but discouraged

Java: no Go To

Classes as types

```
ABMISpreadsheet bmiSpreadsheet = new
ABMISpreadsheet(1.77, 75);
```

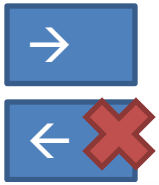
Classes can be used to type variables

```
BMISpreadsheet bmiSpreadsheet = new
ABMISpreadsheet(1.77, 75);
```

Use interfaces instead

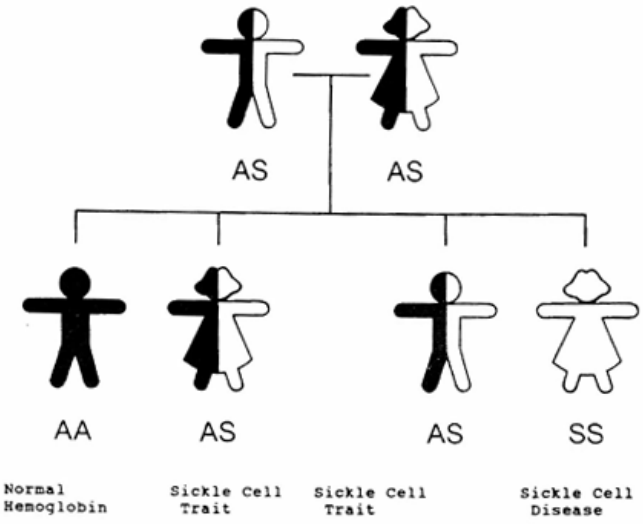
Inheritance vs. (Inter-Class) Code Reusability

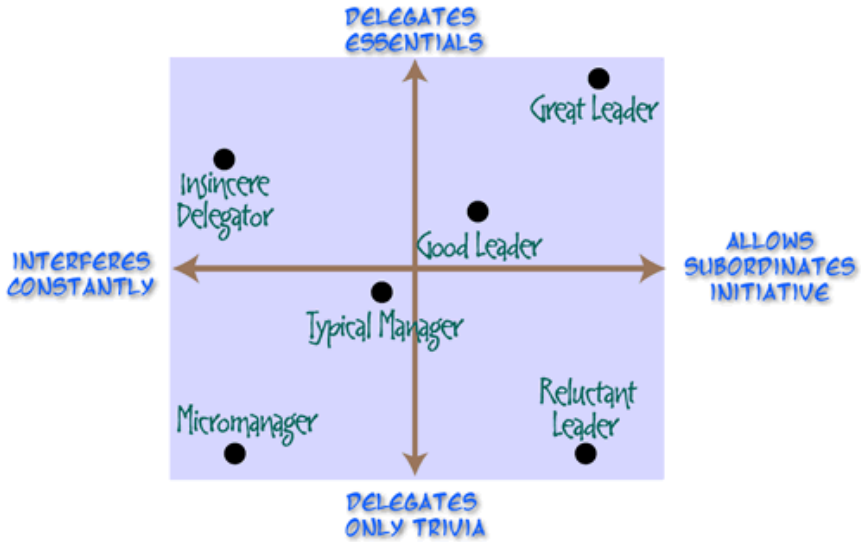
Inheritance (IS-A)



Code Reusability

Delegation (HAS-A)





Can inherit basketball skills

Can pass to someone who has skills

Inheritance requires less work

Delegation can be changed

Delegation Example: PointHistory

```
public interface History<T> {
        public void addElement(T element);
        public T elementAt (int index);
        public int size();
}
```

```
public interface PointHistory extends History<String> {

}
```

```
public interface PointHistory {
        public void addElement (int x, int y);
        public Point elementAt (int index);
        public int size();
}
```

Inheriting APointHistory

```
public class APointHistory extends AHistory<Point>
implements PointHistory{
    public void addElement(int x, int y) {
        addElement(new ACartesianPoint(x,y));
    }
}
```

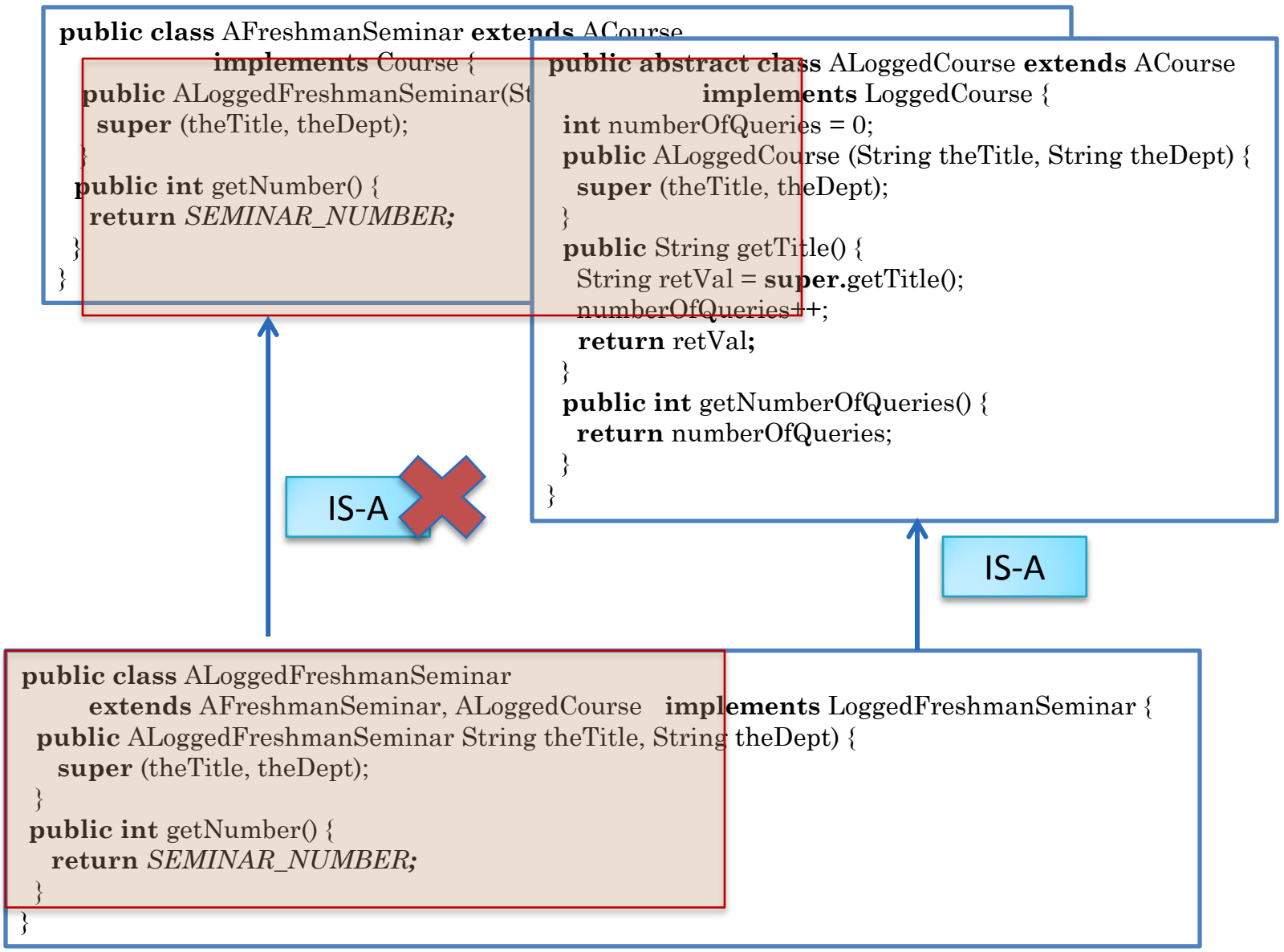
Has both addElement methods!

DELEGATING APOINTHISTORY

```
public class APointHistory implements PointHistory {
        History<Point> contents = new AHistory();
        public void addElement(int x, int y) {
                contents.addElement(new ACartesianPoint(x,y));
        }
        public Point elementAt(int index) {
                return contents.elementAt(index);
        }
        public int size() {
                return contents.size();
        }
}
```

In delegation, class C1 has-a reference to reused class C2, rather than C1 IS-A C2

Multiple Inheritance Problem



DELEGATION

```
public class AFreshmanSeminar extends ACourse
        implements Course {
  public ALoggedFreshmanSemi
    super (theTitle, theDept);
  }
  public int getNumber() {
    return SEMINAR_NUMBER;
  }
}
```

```
public abstract class ALoggedCourse extends ACourse
        implements LoggedCourse {
  int numberOfQueries = 0;
  public ALoggedCourse (String theTitle, String theDept) {
    super (theTitle, theDept);
  }
  public String getTitle() {
    String retVal = super.getTitle();
    numberOfQueries++;
    return retVal;
  }
  public int getNumberOfQueries() {
    return numberOfQueries;
  }
}
```

HAS-A

IS-A

```
public class ALoggedFreshmanSeminar extends ALoggedCourse implements LoggedFreshmanSeminar {
  FreshmanSeminar seminar;
  public ALoggedFreshmanSeminar (String theTitle, String theDept) {
    super (theTitle, theDept);
    seminar= new AFreshmanSeminar(theTitle, theDept,);
  }
  public int getNumber() {
    return  seminar.getNumber();
  }
}
```

Counter

```
public class ACounter implements Counter {
        int counter = 0;
        public void add (int amount) {
                counter += amount;
        }
        public int getValue() {
                return counter;
        }
}
```

Observable Counter

```
public class AnObservableCounter implements ObservableCounter {
        int counter = 0;
        ObserverHistory observers = new AnObserverHistory();
        public void add (int amount) {
                counter += amount;
                notifyObservers();
        }
        public int getValue() {
                return counter;
        }
        public void addObserver(CounterObserver observer) {
                observers.addElement(observer);
                observer.update(this);
        }
        void notifyObservers() {
                for (int observerNum = 0; observerNum < observers.size();
                        observerNum++)
                observers.elementAt(observerNum).update(this);
        }
}
```

Inheriting Observable Counter

```
public class AnObservableCounter extends ACounter implements
ObservableCounter {
        ObserverHistory observers = new AnObserverHistory();
        public void add (int amount) {
                super.add(amount);
                notifyObservers();
        }
        public void addObserver(CounterObserver observer) {
                observers.addElement(observer);
                observer.update(this);
        }
        void notifyObservers() {
                for (int observerNum = 0; observerNum < observers.size();
                        observerNum++)
                observers.elementAt(observerNum).update(this);
        }
}
```

Delegating Observable Counter

```
public class ADelegatingObservableCounter implements ObservableCounter {
        Counter counter ;
        public AnObservableCounter (Counter theCounter) {
                counter = theCounter;
        }
        ObserverHistory observers = new AnObserverHistory();
        public void add (int amount) {
                counter.add(amount);
                notifyObservers();
        }
        public int getValue() {return counter.getValue();}
        public void addObserver(CounterObserver observer) {
                observers.addElement(observer);
                observer.update(this);
        }
        void notifyObservers() {
                for (int observerNum = 0; observerNum < observers.size();
                        observerNum++)
                observers.elementAt(observerNum).update(this);
        }
}
```

Delegation

- Class reusing code HAS-A reference to the Reused class and not IS-A.
- Reusing class is called delegator and reused class is called delegate

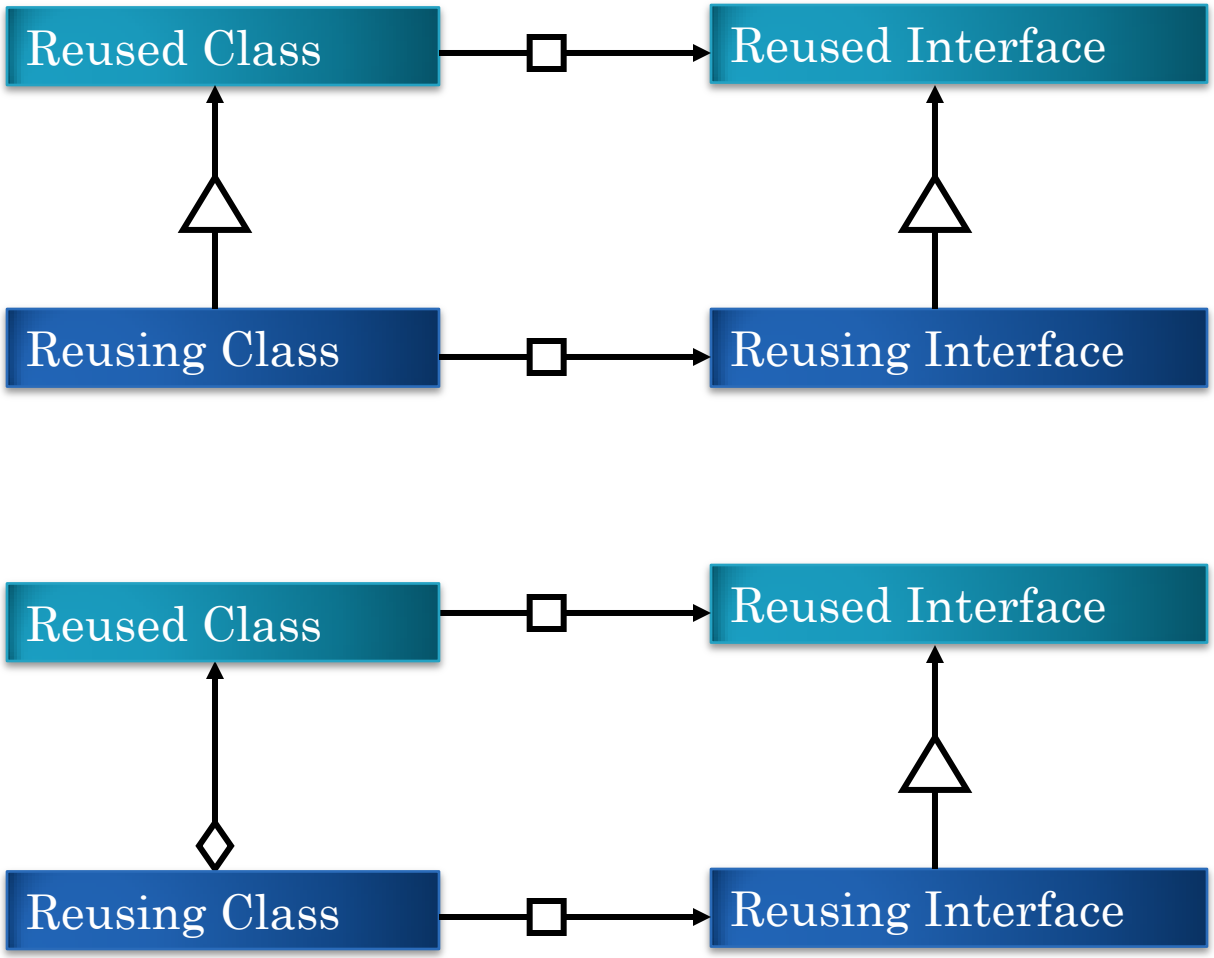
Inheriting Observable Counter

```
public class AnObservableCounter extends ACounter implements
ObservableCounter {
        ObserverHistory observers = new AnObserverHistory();
        public void add (int amount) {
                super.add(amount);
                notifyObservers();
        }
        public void addObserver(CounterObserver observer) {
                observers.addElement(observer);
                observer.update(this);
        }
        void notifyObservers() {
                for (int observerNum = 0; observerNum < observers.size();
                        observerNum++)
                observers.elementAt(observerNum).update(this);
        }
}
```

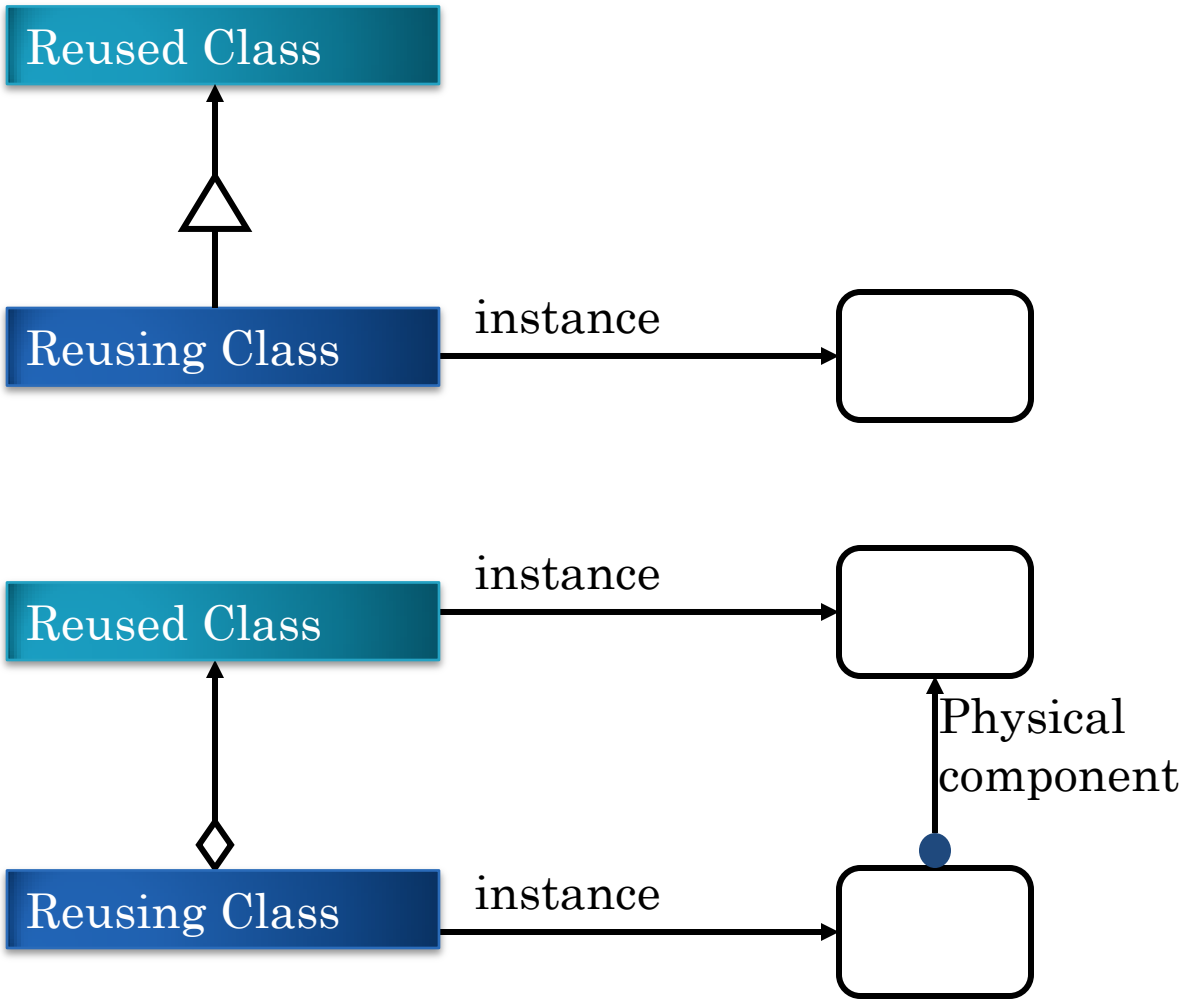
Delegating Observable Counter

```
public class AnObservableCounter implements ObservableCounter {
        Counter counter = new ACounter();
        ObserverHistory observers = new AnObserverHistory();
        public void add (int amount) {
                counter.add(amount);
                notifyObservers();
        }
        public int getValue() {
                return counter.getValue();
        }
        public void addObserver(CounterObserver observer) {
                observers.addElement(observer);
                observer.update(this);
        }
        void notifyObservers() {
                for (int observerNum = 0; observerNum < observers.size();
                        observerNum++)
                observers.elementAt(observerNum).update(this);
        }
}
```

Inheritance vs. Delegation



Inheritance vs. Delegation: Instances



Inheriting Observable Counter

```
public class AnObservableCounter extends ACounter implements
ObservableCounter {
        ObserverHistory observers = new AnObserverHistory();
        public void add (int amount) {
                super.add(amount);
                notifyObservers();
        }
        public void addObserver(CounterObserver observer) {
                observers.addElement(observer);
                observer.update(this);
        }
        void notifyObservers() {
                for (int observerNum = 0; observerNum < observers.size();
                        observerNum++)
                observers.elementAt(observerNum).update(this);
        }
}
```

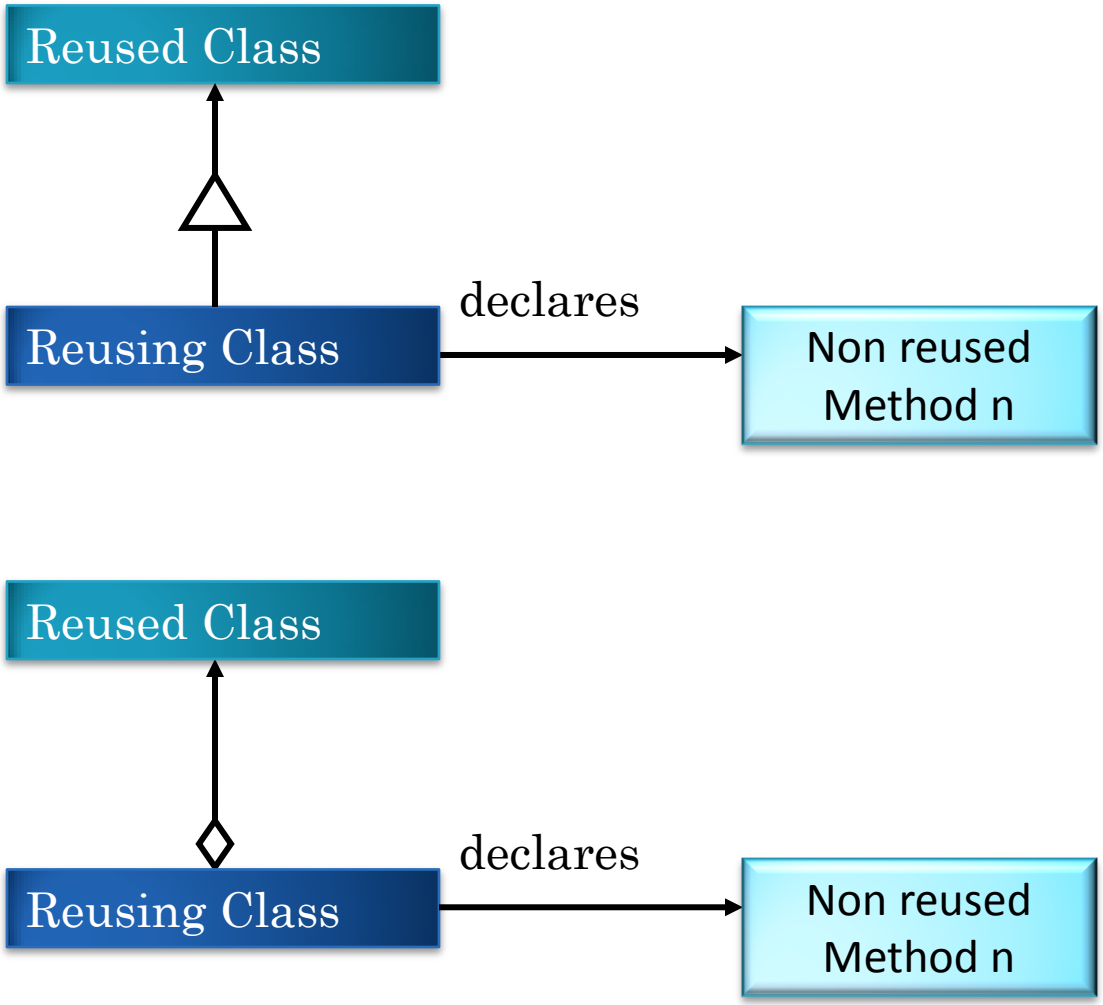
Non-reused methods

Delegating Observable Counter

```
public class AnObservableCounter implements ObservableCounter {
        Counter counter = new ACounter();
        ObserverHistory observers = new AnObserverHistory();
        public void add (int amount) {
                counter.add(amount);
                notifyObservers();
        }
        public int getValue() {
                return counter.getValue();
        }
        public void addObserver(CounterObserver observer) {
                observers.addElement(observer);
                observer.update(this);
        }
        void notifyObservers() {
                for (int observerNum = 0; observerNum < observers.size();
                        observerNum++)
                observers.elementAt(observerNum).update(this);
        }
}
```

Non-reused methods

Inheritance vs. Delegation: Non-Reused Methods (No Difference)



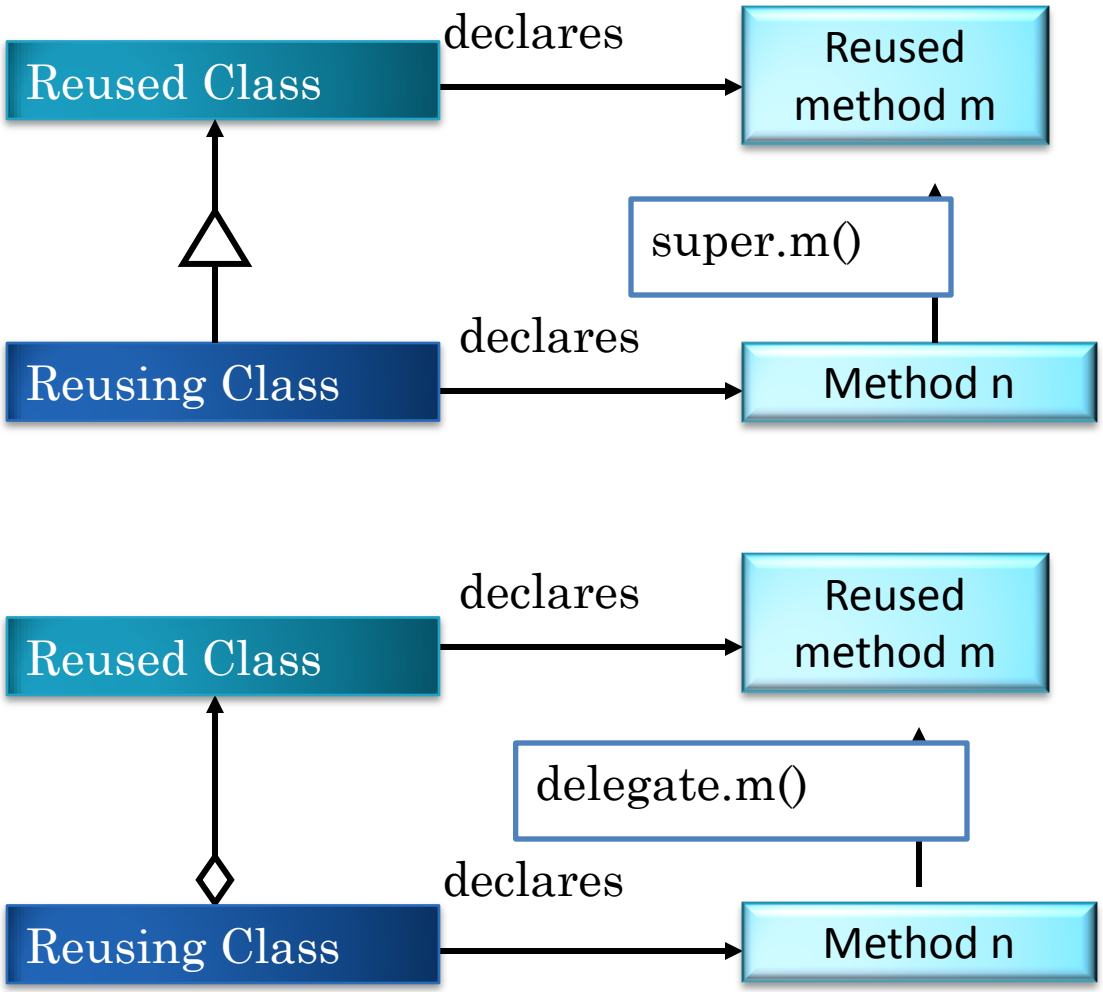
Inheriting Observable Counter

```
public class AnObservableCounter extends ACounter implements
ObservableCounter {
        ObserverHistory observers = new AnObserverHistory();
        public void add (int amount) {
                super.add(amount);
                notifyObservers();
        }
        public void addObserver(CounterObserver observer) {
                observers.addElement(observer);
                observer.update(this);
        }
        void notifyObservers() {
                for (int observerNum = 0; observerNum < observers.size();
                        observerNum++)
                observers.elementAt(observerNum).update(this);
        }
}
```

Delegating Observable Counter

```
public class AnObservableCounter implements ObservableCounter {
        Counter counter = new ACounter();
        ObserverHistory observers = new AnObserverHistory();
        public void add (int amount) {
                counter.add(amount);
                notifyObservers();
        }
        public int getValue() {
                return counter.getValue();
        }
        public void addObserver(CounterObserver observer) {
                observers.addElement(observer);
                observer.update(this);
        }
        void notifyObservers() {
                for (int observerNum = 0; observerNum < observers.size();
                        observerNum++)
                observers.elementAt(observerNum).update(this);
        }
}
```

Inheritance vs. Delegation: Super calls



Delegating Observable Counter

```
public class AnObservableCounter implements ObservableCounter {
        Counter counter = new ACounter();
        ObserverHistory observers = new AnObserverHistory();
        public void add (int amount) {
                counter.add(amount);
                notifyObservers();
        }
        public int getValue() {
                return counter.getValue();
        }
        public void addObserver(CounterObserver observer) {
                observers.addElement(observer);
                observer.update(this);
        }
        void notifyObservers() {
                for (int observerNum = 0; observerNum < observers.size();
                        observerNum++)
                observers.elementAt(observerNum).update(this);
        }
}
```

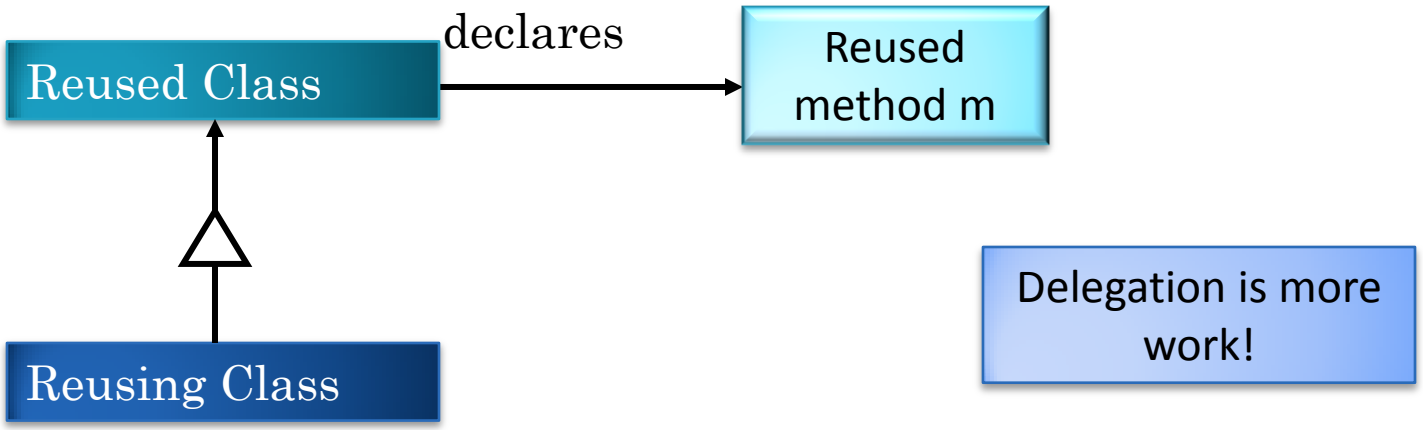
Must declare stub for getValue()

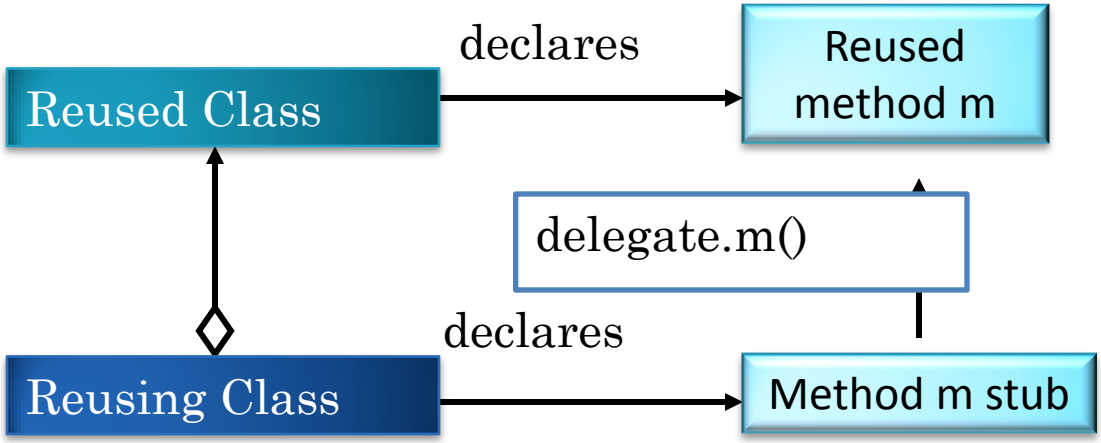
Inheriting Observable Counter

```
public class AnObservableCounter extends ACounter implements
ObservableCounter {
        ObserverHistory observers = new AnObserverHistory();
        public void add (int amount) {
                super.add(amount);
                notifyObservers();
        }
        public void addObserver(CounterObserver observer) {
                observers.addElement(observer);
                observer.update(this);
        }
        void notifyObservers() {
                for (int observerNum = 0; observerNum < observers.size();
                        observerNum++)
                observers.elementAt(observerNum).update(this);
        }
}
```

getValue() inherited without overriding

Inheritance vs. Delegation: Stubs





Declaring Abstract methods

```
public abstract class AnAbstractPointHistory implements PointHistory {
  public final int MAX_SIZE = 50;
  protected Point[] contents = new Point[MAX_SIZE];
  protected int size = 0;
  public int size() {
    return size;
  }
  public Point elementAt (int index) {
    return contents[index];
  }
  protected boolean isFull() {
    return size == MAX_SIZE;
  }
  public void addElement(int x, int y) {
    if (isFull())
      System.out.println("Adding item to a full history");
    else {
      Point p = createPoint(x, y);
      contents[size] = p;
       size++;
     }
  }
   protected abstract Point createPoint(int x, int y);
}
```

Inheriting Abstract Methods

```
public class ACartesianPointHistory extends AnAbstractPointHistory{
  protected Point createPoint(int x, int y) {
    return new ACartesianPoint(x, y);
  }
}
```

Inheriting Abstract Methods

```
public class APolarPointHistory extends AnAbstractPointHistory{
  protected Point createPoint(int x, int y) {
    return new APolarPoint(x, y);
  }
}
```

Equivalent Delegate Class?

```
public abstract class AnAbstractPointHistory implements PointHistory {
  public final int MAX_SIZE = 50;
  protected Point[] contents = new Point[MAX_SIZE];
  protected int size = 0;
  public int size() {
    return size;
  }
  public Point elementAt (int index) {
    return contents[index];
  }
  protected boolean isFull() {
    return size == MAX_SIZE;
  }
  public void addElement(int x, int y) {
    if (isFull())
      System.out.println("Adding item to a full history");
    else {
      Point p = createPoint(x, y);
      contents[size] = p;
       size++;
     }
  }
  protected abstract Point createPoint(int x, int y);
}
```

Equivalent Delegate Class

```
public class ADelegatePointHistory implements PointHistory {
  public final int MAX_SIZE = 50;
  protected Point[] contents = new Point[MAX_SIZE];
  protected int size = 0;
  DelegatingPointHistory delegator;
  public ADelegatePointHistory(DelegatingPointHistory theDelegator) {
    delegator = theDelegator;
  }
  public int size() {return size;}
  public Point elementAt (int index) {return contents[index];}
  protected boolean isFull() {return size == MAX_SIZE;}
  public void addElement(int x, int y) {
    if (isFull())
      System.out.println("Adding item to a full history");
    else {
      Point p = delegator.createPoint(x, y);
      contents[size] = p;
      size++;
    }
  }
}
```

Callbacks invoked on delagator

Delegate class passed reference to delegator

Delegate class is concrete, hece implementss interface

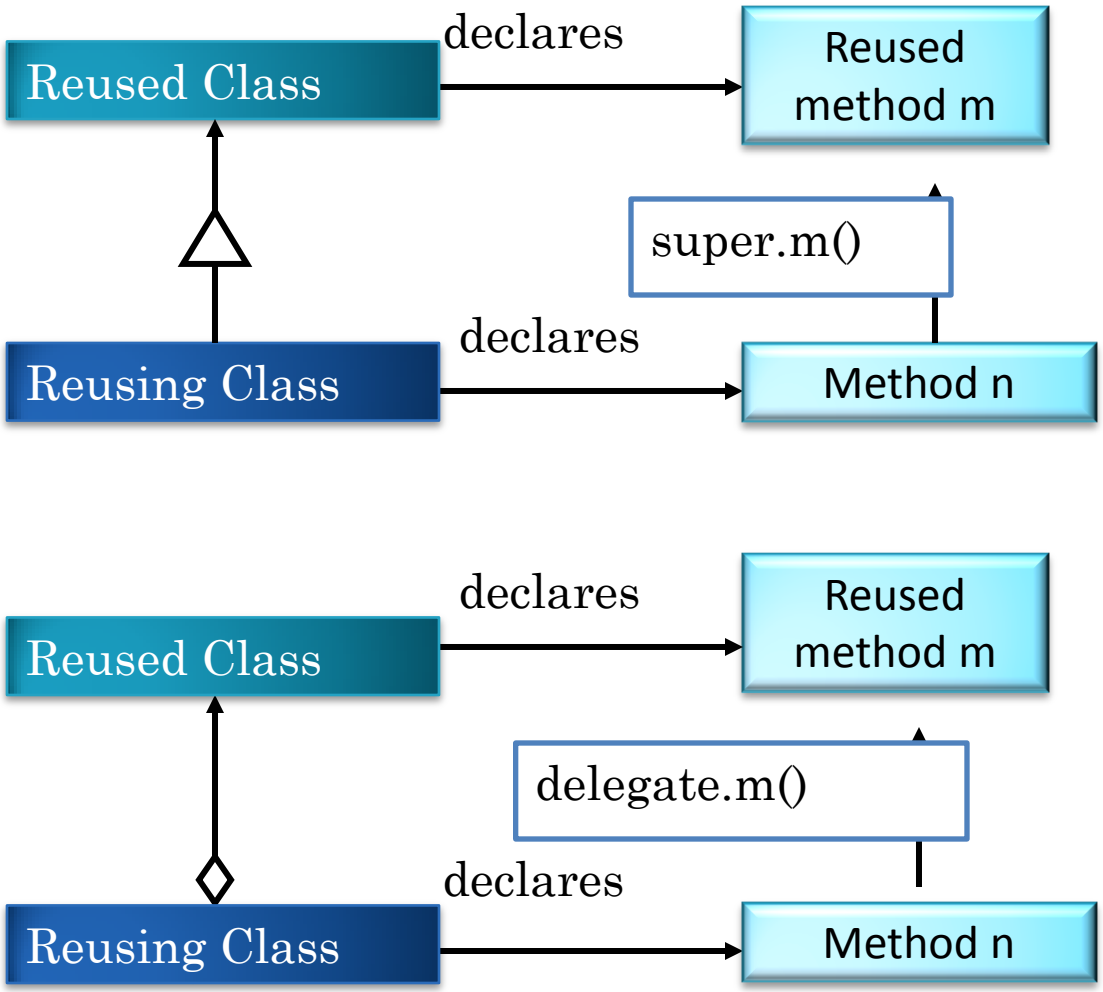
Equivalent Concrete Class?

```
public class ACartesianPointHistory
                extends AnAbstractPointHistory{
  protected Point createPoint(int x, int y) {
     return new ACartesianPoint(x, y);
   }
}
```

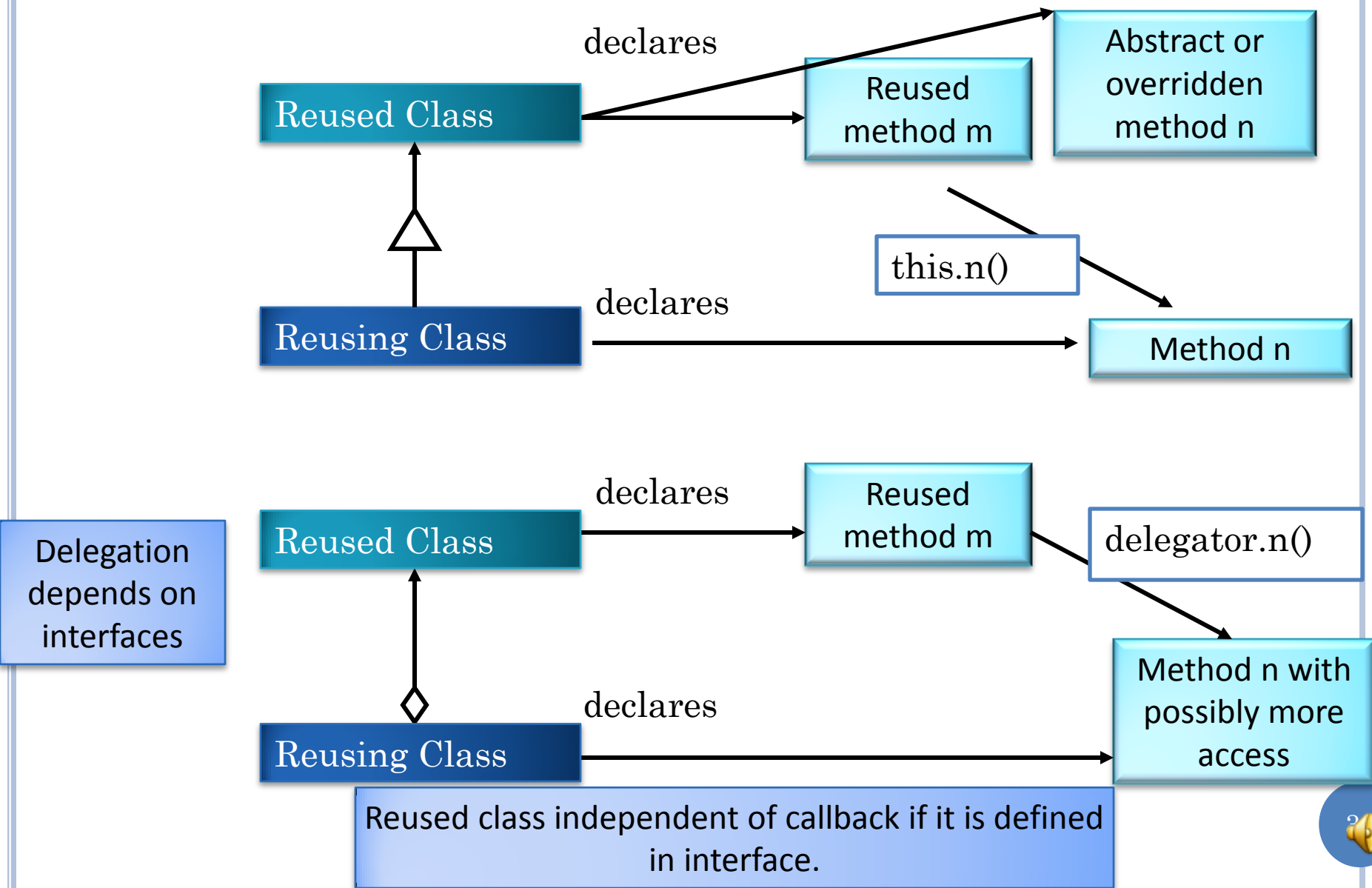
Callbacks Must Be Public

```
public class ADelegatingCartesianPointHistory
                implements DelegatingPointHistory{
  PointHistory delegate;
  public void addElement(int x, int y) {
    delegate.addElement(x, y);
  }
  public Point elementAt(int index) {
    return delegate.elementAt(index);
  }
  public int size() {
    return delegate.size();
  }
  public Point createPoint(int x, int y) {
    return new ACartesianPoint(x, y);
  }
}
```

Inheritance vs. Delegation: Calls



Inheritance vs. Delegation: Callbacks



Calls vs. Callbacks

- Calls
  - calls from reusing class to reused class
- Callbacks
  - calls from reused class to reusing class.
  - not to implement a symbiotic relationship
    - Delegate has no idea what the callback does, is independent of it
    - Model notification is example of callback
  - done to service calls

Access problems

- Occur with callbacks.
- And if reusing class accesses variables of reused class
- Must violate principle of least privilege for methods

Inheriting String Database

```
public class AStringDatabase extends AStringHistory implements
StringDatabase {
        public void deleteElement (String element) {
                shiftUp(indexOf(element));
        }
        public int indexOf (String element) {
                int index = 0;
                while ((index < size) && !element.equals(contents[index]))
                        index++;
                return index;
        }
        void shiftUp (int startIndex) {…}
        public boolean member(String element) {
                return indexOf (element) < size;
        }
        public void clear() {size = 0;}
}
```

Delegating String Database

```
public class ADelegatingStringDatabase implements StringDatabase {
  AStringHistoryDelegate stringHistory = new AStringHistoryDelegate();
  public int indexOf (String element) {
    int index = 0;
    while ((index < stringHistory.size()) &&
         !element.equals(stringHistory.elementAt(index)))
      index++;
    return index;
  }
  void shiftUp(int startIndex) {
    for (int index = startIndex; index + 1 < stringHistory.size(); index++) {
      stringHistory.contents[index] = stringHistory.elementAt(index + 1);
    }
    stringHistory.size--;
  }
  public void removeElement(String element) {
    shiftUp(indexOf(element));
  }
  public boolean member(String element) {
    return indexOf (element) < stringHistory.size();
  }
  public void clear() {
    stringHistory.size = 0;
  }
```

Class as type and accessing non public variables in same package

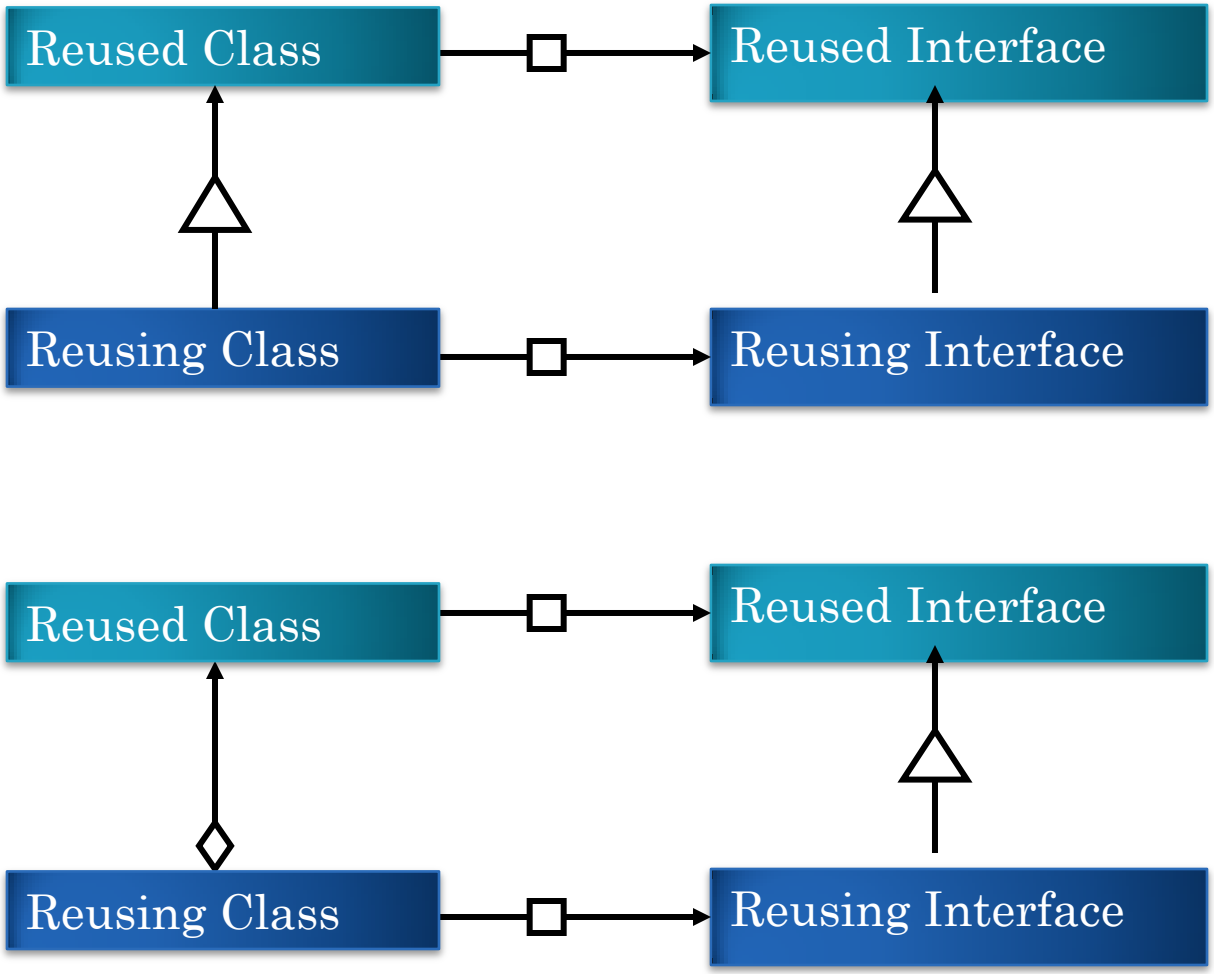
Access issues

- Callbacks methods
  - must be given public access if interface types the delegator.
  - can be given protected or default access
  - but requires class types, delegator and delegator class in same package
- May need to give access to reused class variables also
  - through public methods
  - or putting delegate and delegator in same package, giving them protected/default access and using class as type

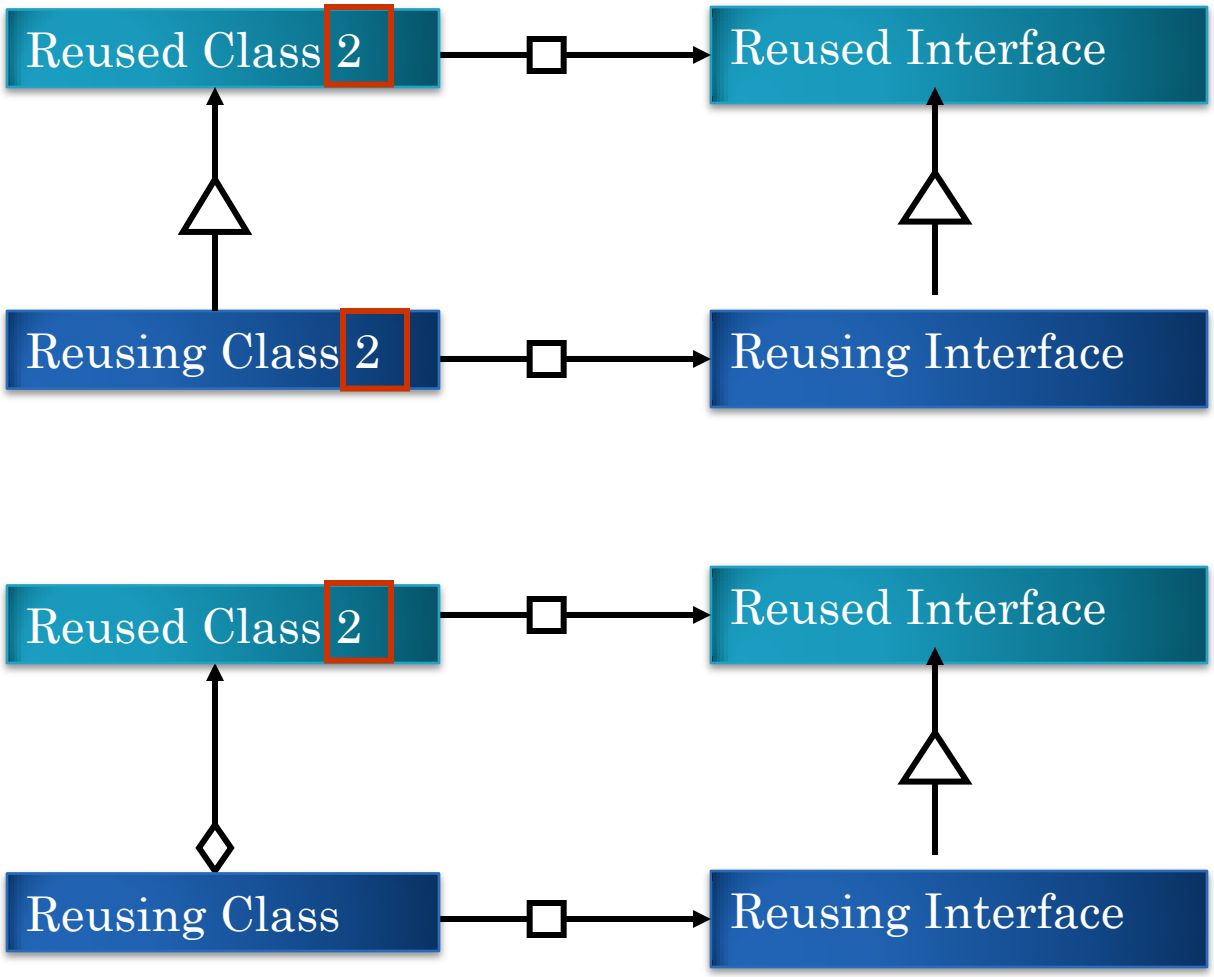
Cons of Delegation

- Need to instantiate multiple classes.
- Need to compose instances.
  - Boombox vs assembled stereo
- Need to define stub methods
- Access problems
- Need to define special interfaces for "non standalone" class.

Pros of Delegation



Substituting Reused Class



Counter

```
public class AResettingCounter implements Counter {
  int counter = 0;
  public void add (int amount) {
  if ((counter  - amount) >= Integer.MAX_VALUE)
    counter = 0;
  else
    counter += amount;
  }
  public int getValue() {
    return counter;
  }
}
```

Inheriting Observable Counter

```
public class AnInheritingObservableResettingCounter extends
        AResettingCounter implements ObservableCounter {
  ObserverHistory observers = new AnObserverHistory();
  public void add (int amount) {
    super.add(amount);
    notifyObservers();
  }
  public void addObserver(CounterObserver observer) {
    observers.addElement(observer);
    observer.update(this);
  }
  void notifyObservers() {
    for (int observerNum = 0; observerNum < observers.size();
          observerNum++)
      observers.elementAt(observerNum).update(this);
    }
  }
}
```

Delegating Observable Counter

```
public class ADelegatingObservableCounter implements ObservableCounter {
        Counter counter ;
        public AnObservableCounter (Counter theCounter) {
                counter = theCounter;
        }
        ObserverHistory observers = new AnObserver
        public void add (int amount) {
                counter.add(amount);
                notifyObservers();
        }
        public int getValue() {return counter.getVa
        public void addObserver(CounterObserver
                observers.addElement(observer);
                observer.update(this);
        }
        void notifyObservers() {
                for (int observerNum = 0; observerNum < observers.size();
                        observerNum++)
                observers.elementAt(observerNum).update(this);
        }
}
```

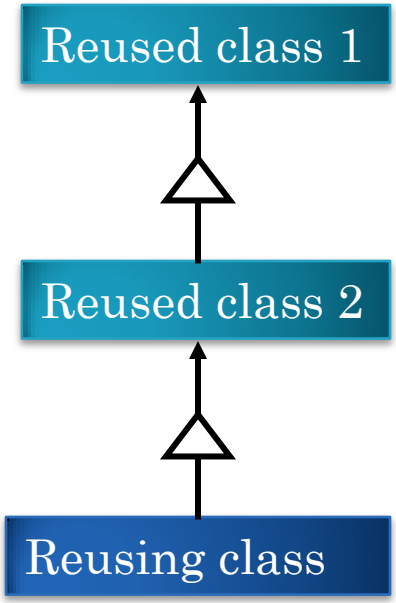
Can be ACounter or AResettingCounter

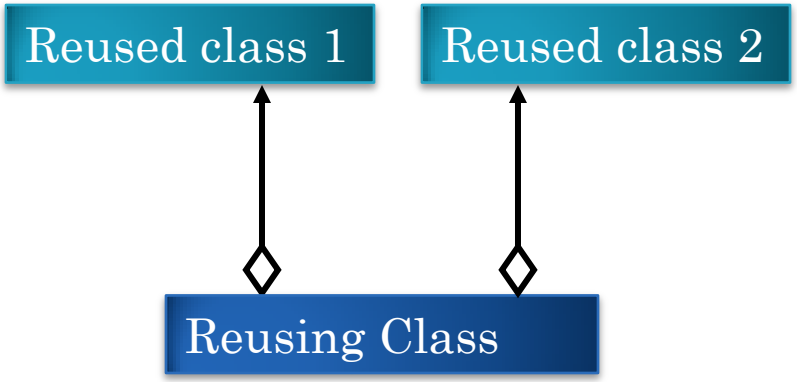
Delegation goes hand in hand with interfaces

Substituting Reused Class

- Another implementation of reused interface
  - in inheritance requires another implementation of reusing class that duplicates methods of original reusing class.
  - In delegation simply requires a different object to be assigned to the delegate variable

Multiple Reused Classes





- Inheritance
  - All reused classes must be in linear superclass chain
  - Java supports single superclass
- Delegation
  - Can have references to multiple classes

Counter

```
public class ACounter implements Counter {
        int counter = 0;
        public void add (int amount) {
                counter += amount;
        }
        public int getValue() {
                return counter;
        }
}
```

Controller with Console View

```
public class ACounterWithConsoleView  extends ACounter {
        void appendToConsole(int counterValue) {
                System.out.println("Counter: " + counterValue);
        }
        public void add (int amount) {
                super.add(amount);
                appendToConsole(getValue());
        }
}
```

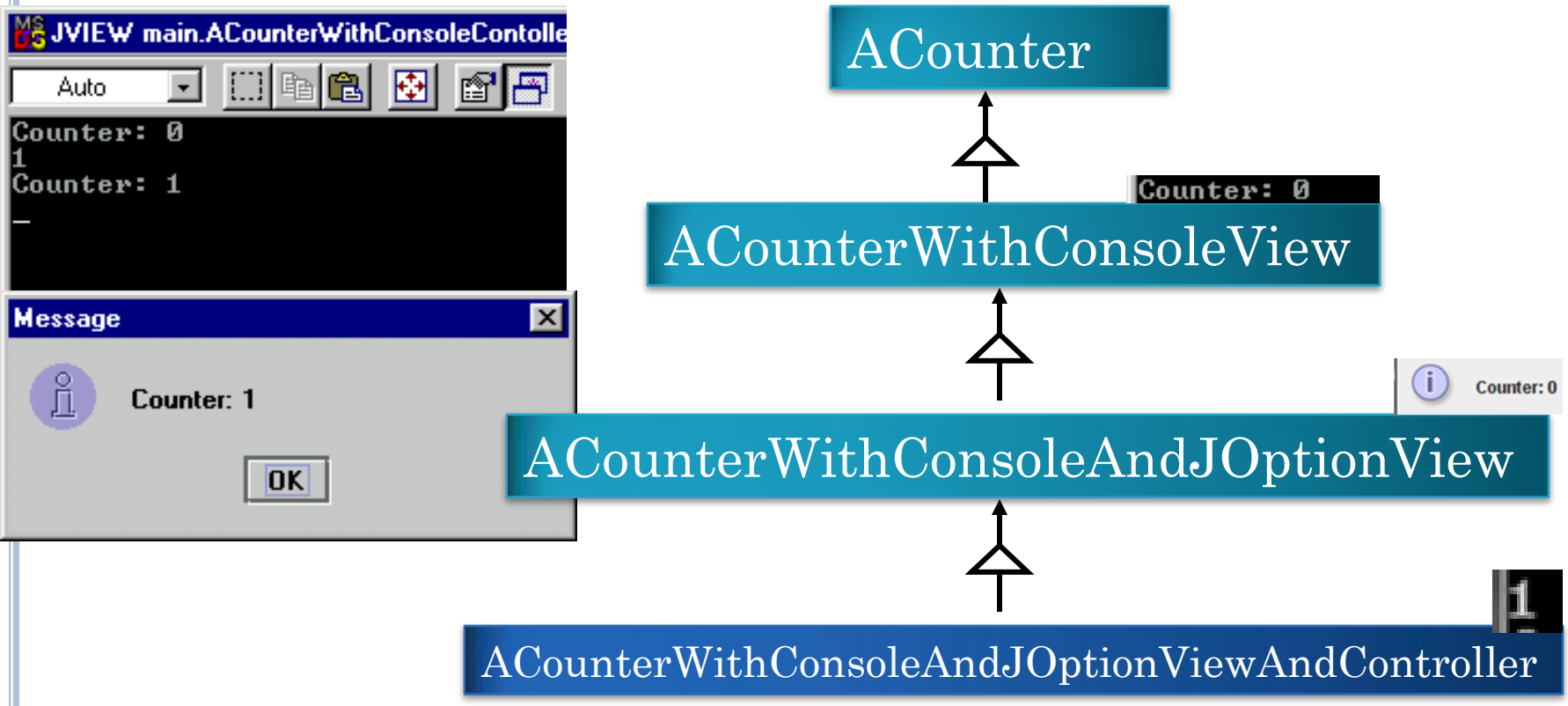
Controller with Console and JOption View

```
import javax.swing.JOptionPane;
public class ACounterWithConsoleAndJOptionView  extends
ACounterWithConsoleView {
        void displayMessage(int counterValue) {
                JOptionPane.showMessageDialog(null, "Counter: " +
counterValue);
        }
        public void add (int amount) {
                super.add(amount);
                displayMessage(getValue());
        }
}
```

Controller with Console and JOption View and Controller

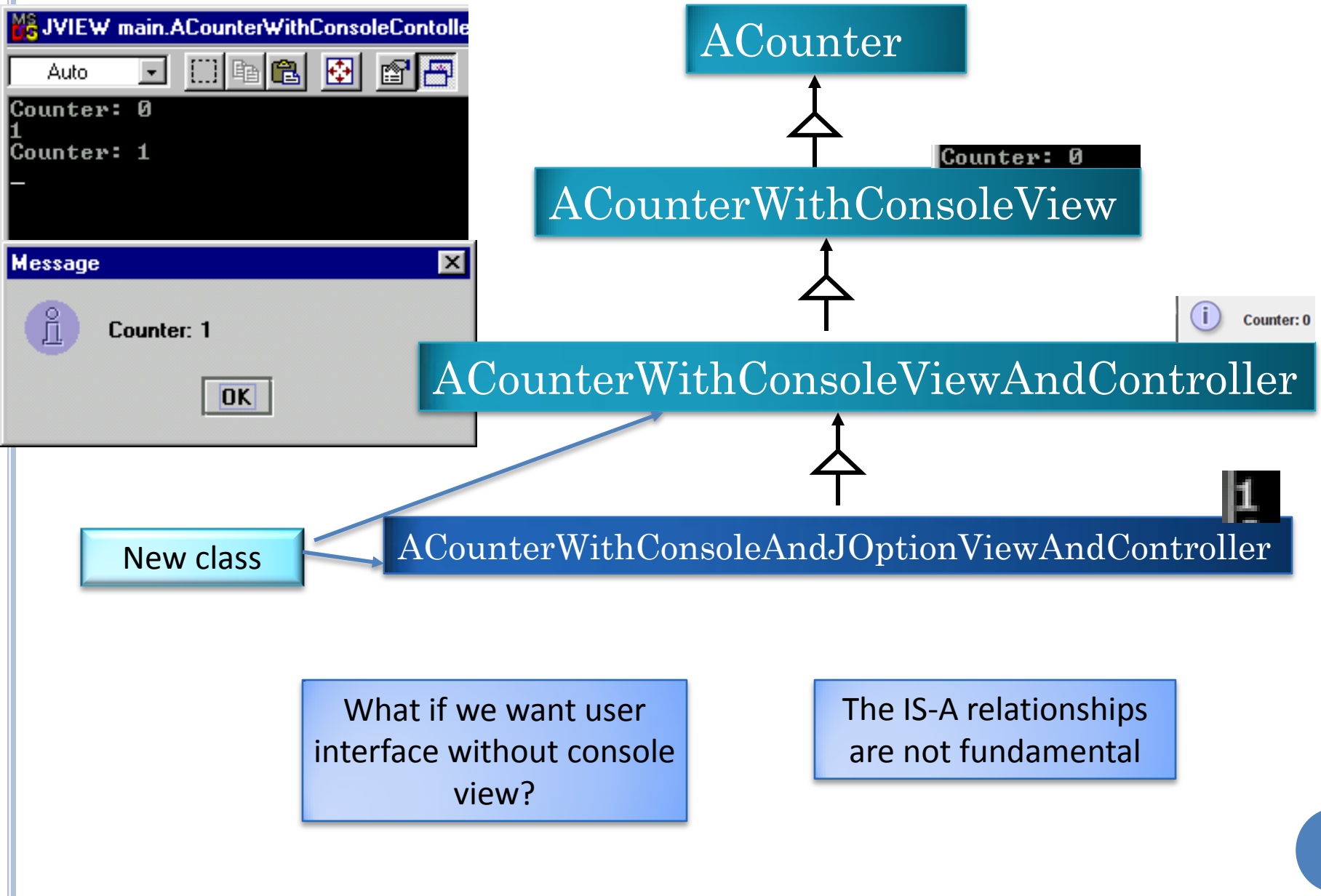
```
public class ACounterWithConsoleAndJOptionViewAndController
                    extends ACounterWithConsoleAndJOptionView
                    implements CounterWithController {
          public void processInput() {
                    while (true) {
                              int nextInput = Console.readInt();
                              if (nextInput == 0) return;
                              add(nextInput);
                    }
          }
}
```

Inheritance-based MVC



What if we want user interface without JOptionView?

Another Inheritance-based MVC



What if we want user interface without console view?

The IS-A relationships are not fundamental

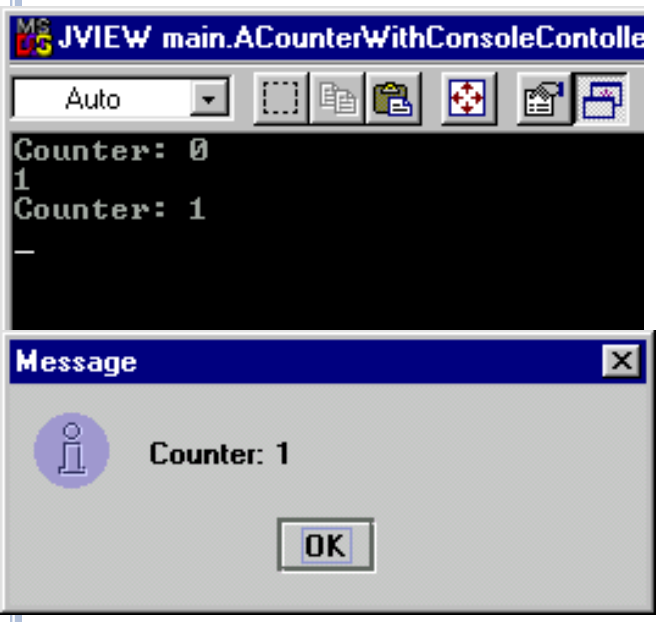
Controller with Console and View and Controller

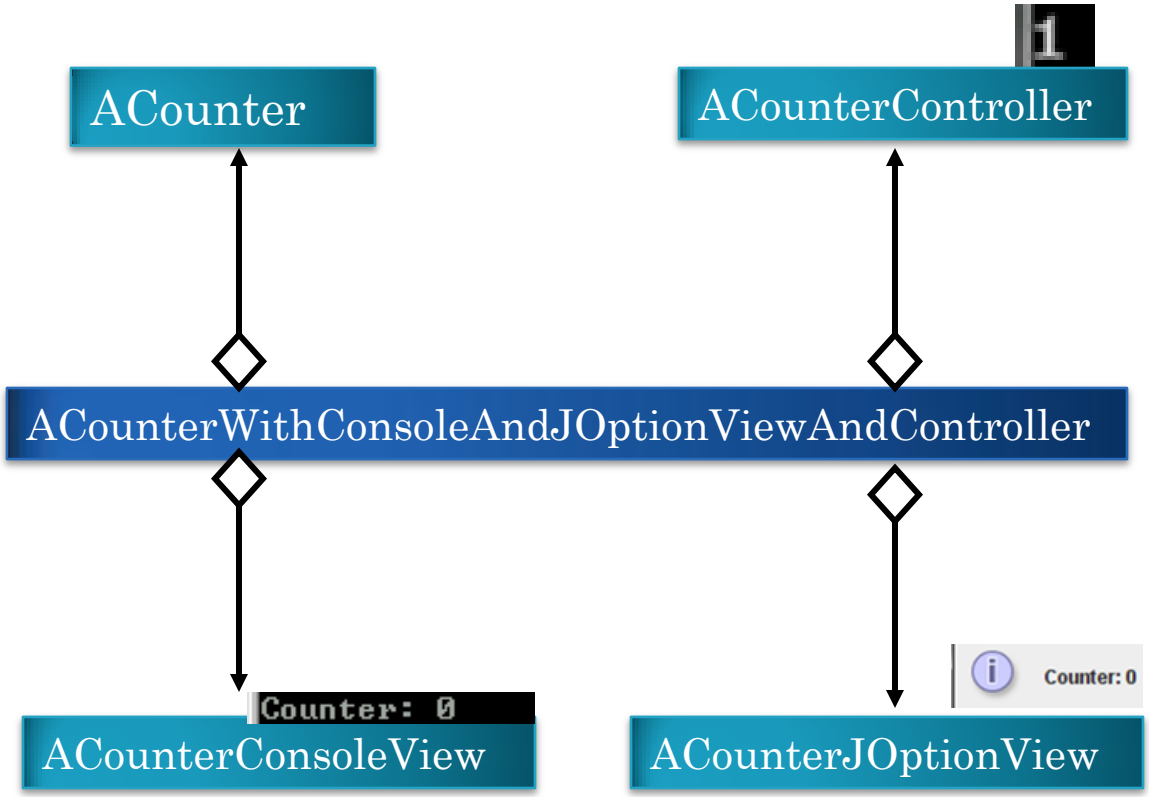
```
public class ACounterWithConsoleViewAndController extends
ACounterWithConsoleView  implements CounterWithController {
        public void processInput() {
                while (true) {
                        int nextInput = Console.readInt();
                        if (nextInput == 0) return;
                        add(nextInput);
                }
        }
```

```
public class ACounterWithConsoleAndJOptionViewAndController extends
ACounterWithConsoleAndJOptionView  implements
CounterWithConsoleViewAndController {
        public void processInput() {
                while (true) {
                        int nextInput = Console.readInt();
                        if (nextInput == 0) return;
                        add(nextInput);
                }
        }
}
```

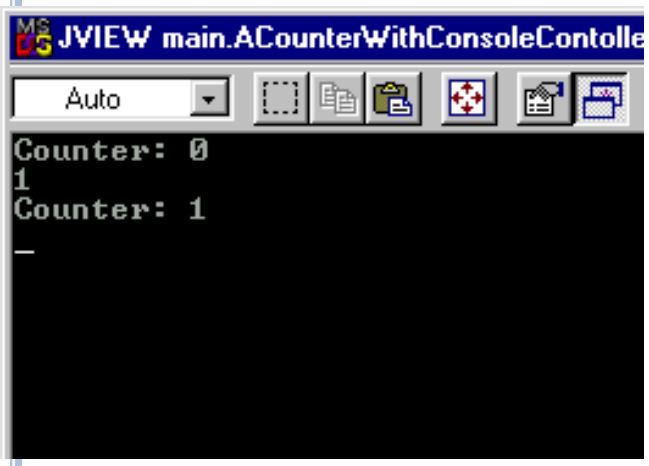
Only difference between it and previous inheriting controller but must create a new class!

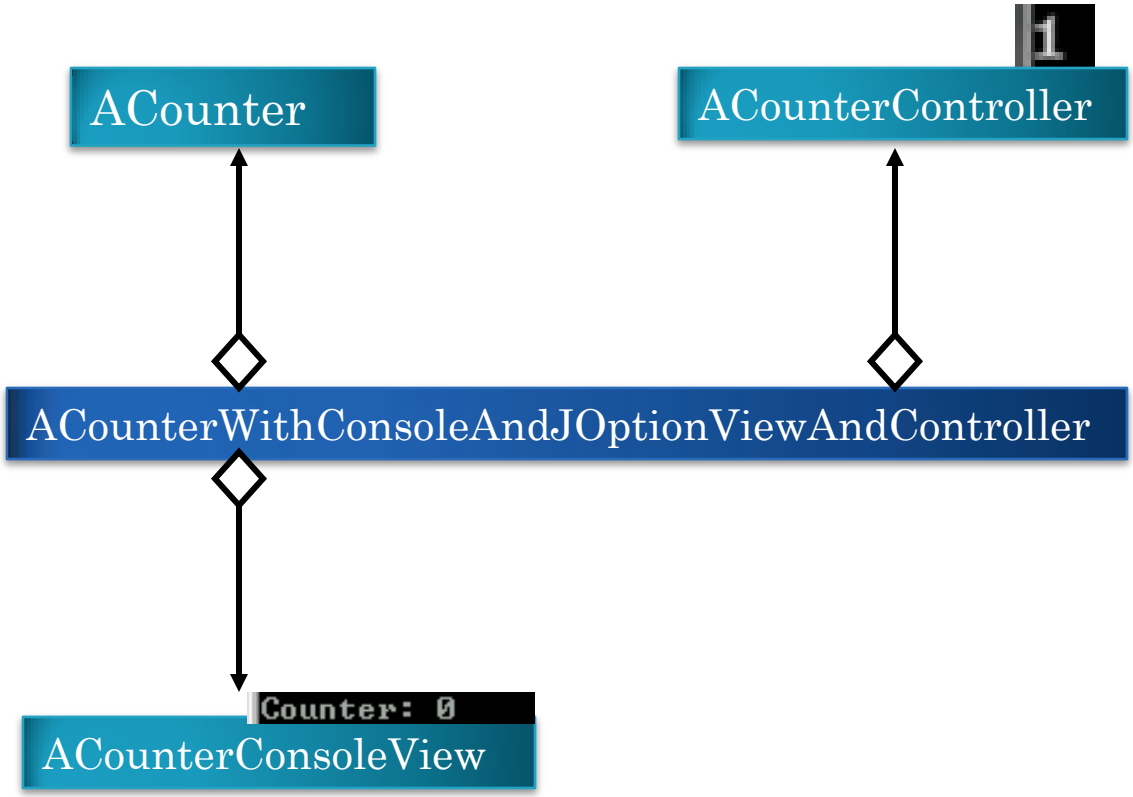
Delegation Based MVC



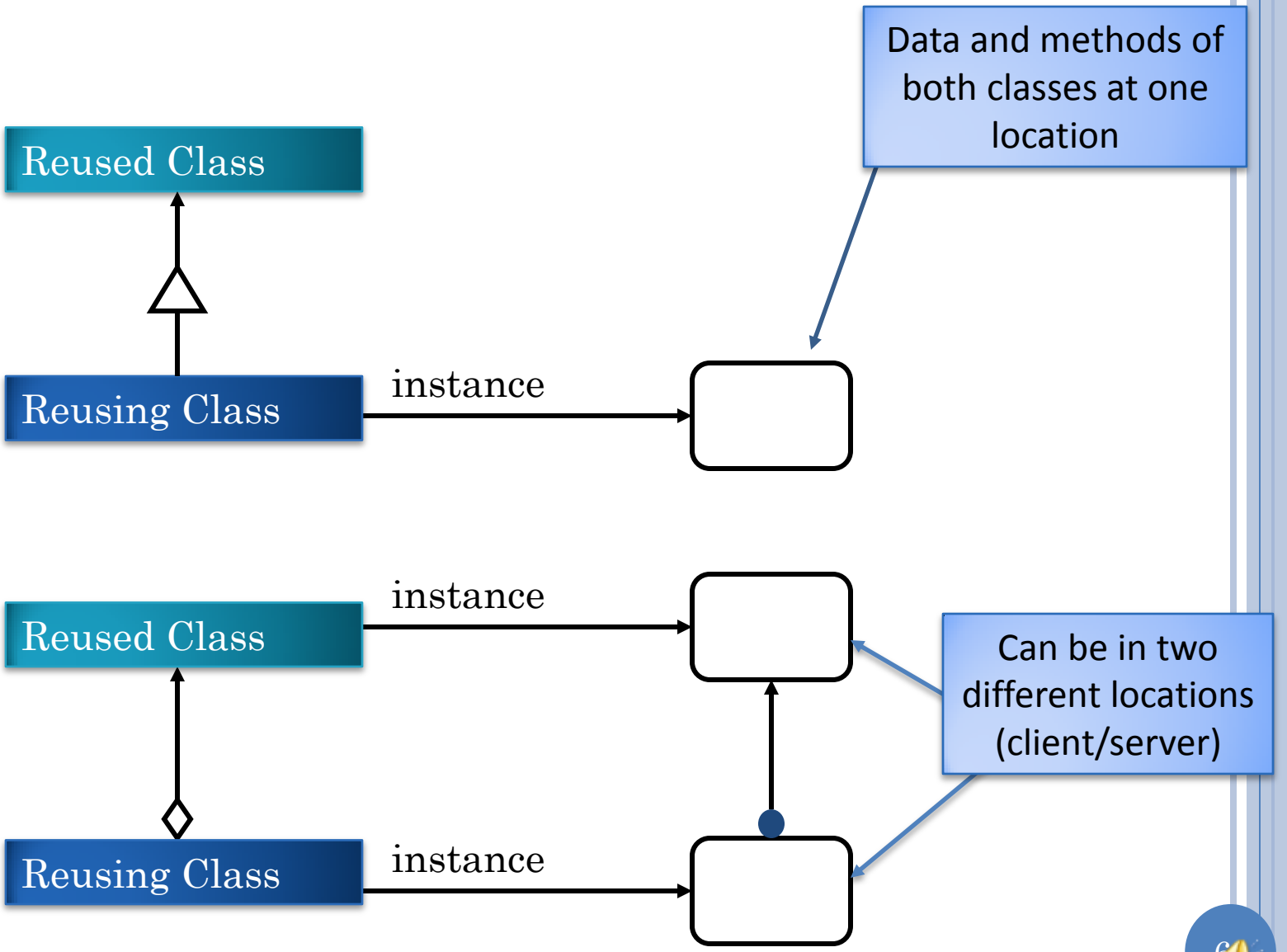


Omitting JOptionView





Inheritance vs. Delegation: Distribution



Pros and Cons of Delegation

- Can change reused class without changing reusing class.
- Multiple reusing classes can share reused class variables simultaneously.
- Variables and methods of reusing and reused class can be on separate computers.
- Works in single inheritance languages.
- Reusing class does not have to support all methods of reused classes.
- Reusing class cannot automatically be substituted for reused class.

- Need to define special interfaces for non standalone class.
- Need to instantiate multiple classes.
- Need to compose instances.
- Need to define stub methods

Need to weigh pros and cons
When benefits of delegation don't apply use inheritance
That is why inheritance exists.

Inheritance is a bad idea

- If reusing class has "with" in its name:
  - ACounterWithConsoleView
- If more than one way to do inheritance
  - Controller can be subclass of view
  - Or vice versa
- If some of the inherited methods are not used:
  - AStringHistory **extends** Vector
    - Adds methods to add and access string elements rather than object elements
    - addString(), stringAt()
  - does not use removeElement() etc
- If reusing class should not be used wherever shared class is used
  - even if all methods are reused.

Inheritance is a bad idea

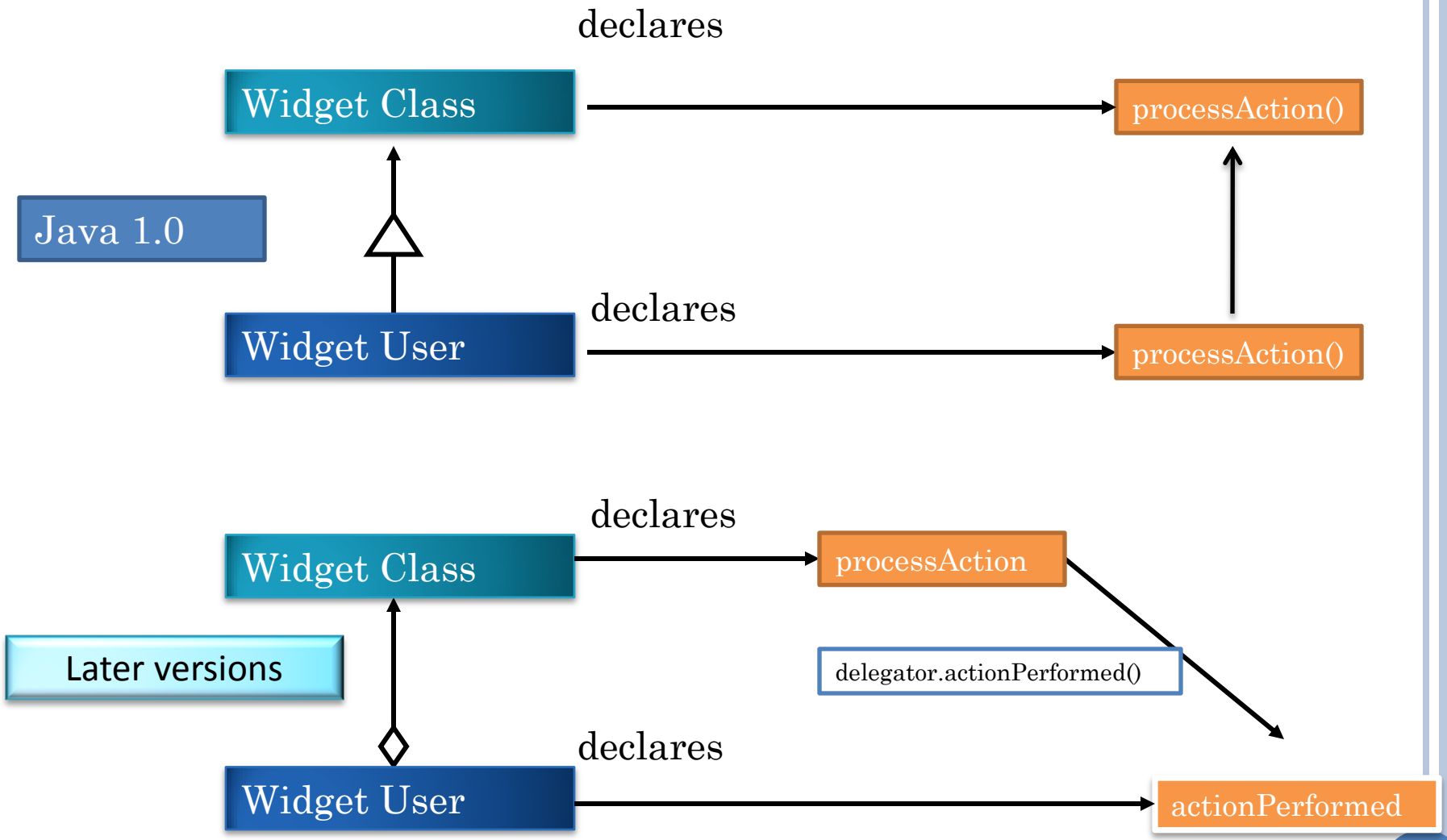

```
public class ASquareHAS_A_Point implements Square {
  Point center;
  int sideLength;
  public ASquareHAS_A_Point (Point theCenter,
                             int theSideLength) {
    center = theCenter;
    sideLength = theSideLength;
  }
  public int getX() {return center.getX();}
    public int getY() {return center.getY();}
    public double getRadius() {return center.getRadius();}
    public double getAngle() {return center.getAngle();}
    public int getSideLength() {return sideLength;}
  }
```

```
public class ASquareIS_A_Point extends ACartesianPoint {
  int sideLength;
  public ASquareIS_A_Point(int theX, int theY,
         int theSideLength) {
    super(theX, theY);
    sideLength = theSideLength;
  }
  public int getSideLength() {return sideLength;}
}
```

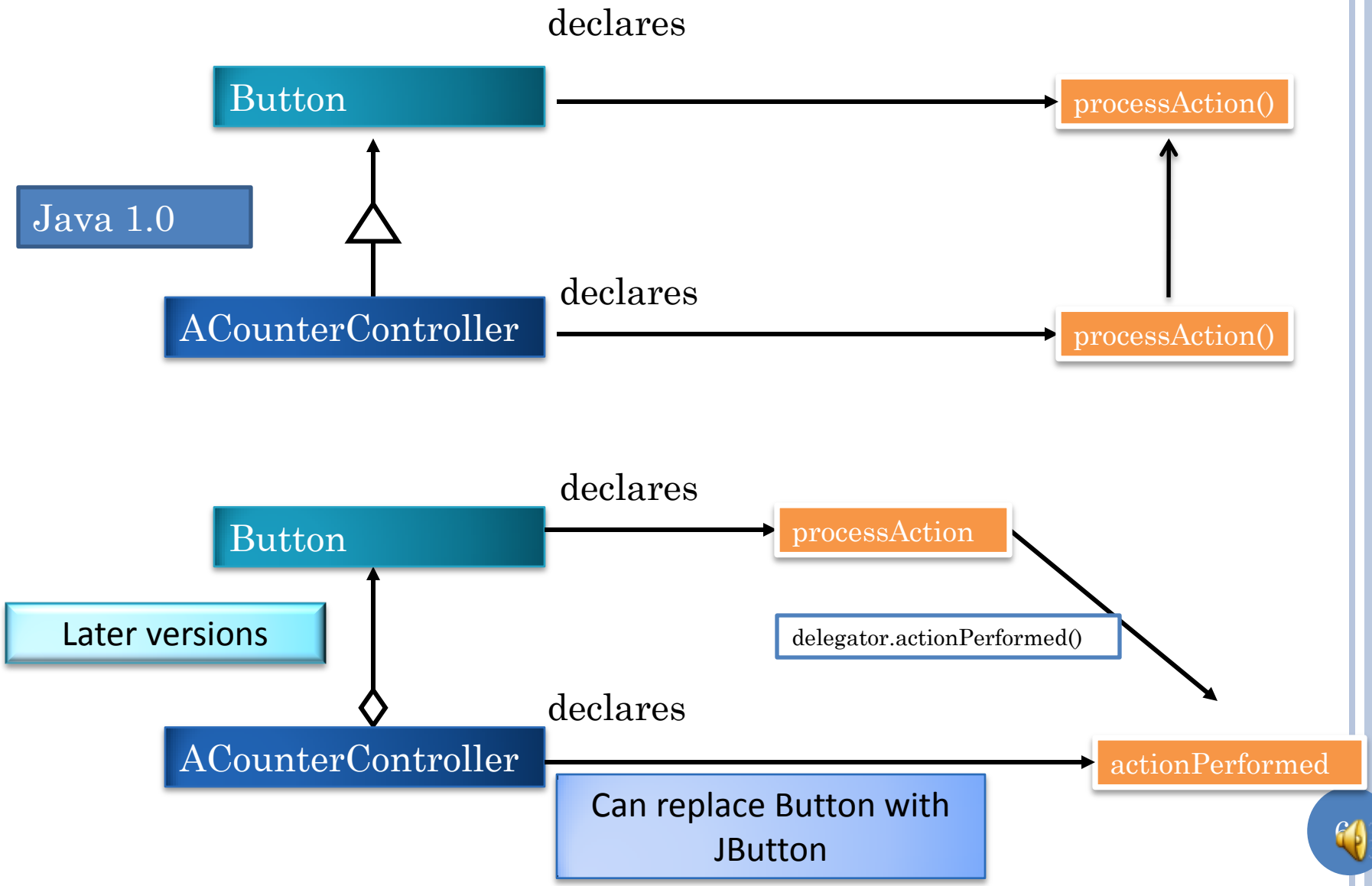
ASquare IS-NOT-A ACartesianPoint!

Cannot use APolarPoint without redoing ASquare!

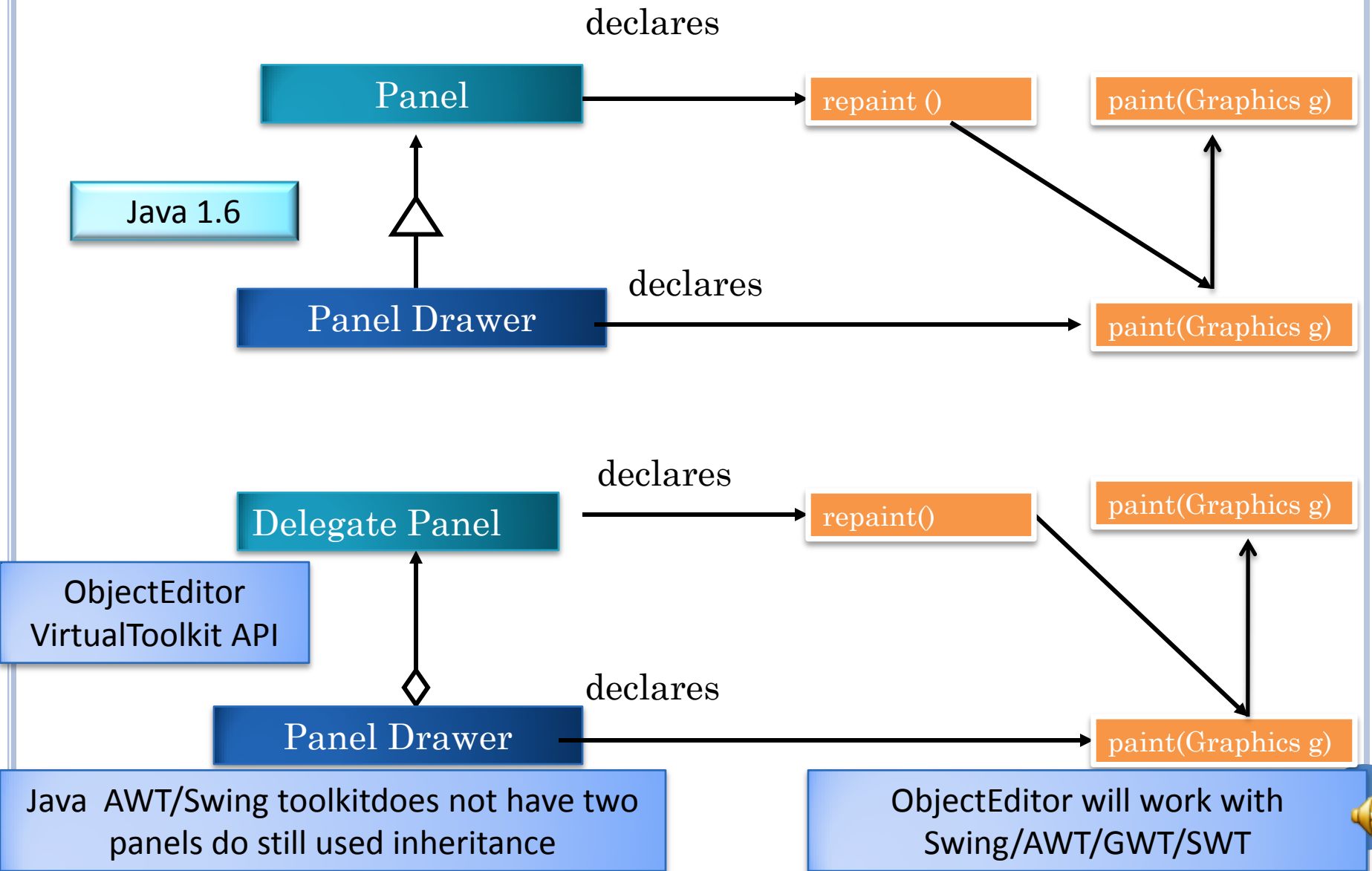
Inheritance vs. Delegation in Popular Software: Toolkits



Inheritance vs. Delegation in Popular Software: Toolkits



Inheritance vs. Delegation in Popular Software: Toolkits



Inheritance vs. Delegation in Popular Software

- Main problem with inheritance based widgets
  - Widget users could not be subclass of any other class because of multiple inheritance
- Multiple objects can receive events from same widget.
- Widget implementation can be switched
- Observer concept supports both approaches
  - Inherited Observer class vs. Delegated PropertyChangeSupport
- Threads support both approaches
  - Inheritable Thread class vs. Implemented Runnable Interface