COMP 303 - Lecture Notes for Week 6 - Inheritance and Abstract Classes

- Slides edited from, Object-Oriented Design Patterns, by Cay S. Horstmann
- Original slides available from: http://www.horstmann.com/design_and_patterns.html
- Modifications made by Laurie Hendren, McGill University
- Topics this week:
  - Inheritance and Abstract Classes
  - Automatic Formatting of Java code (jalopy)
  - Obfuscating Java code (yguard)

Chapter Topics

- The Concept of Inheritance
- Graphics Programming with Inheritance
- Abstract Classes
- The TEMPLATE METHOD Pattern
- Protected Interfaces
- The Hierarchy of Swing Components
- The Hierarchy of Standard Geometrical Shapes
- The Hierarchy of Exception Classes
- When Not to Use Inheritance

Modeling Specialization

- Start with simple Employee class
  ```java
  public class Employee {
    public Employee(String aName) {
      name = aName;
    }
    public void setSalary(double aSalary) {
      salary = aSalary;
    }
    public String getName() {
      return name;
    }
    public double getSalary() {
      return salary;
    }
    private String name;
    private double salary;
  }
  ```
- Manager is a subclass
  ```java
  public class Manager extends Employee {
    public Manager(String aName) {
      super(aName);
    }
    public void setBonus(double aBonus) {
      bonus = aBonus;
    }
    public double getSalary() {
      return salary + bonus;
    }
    private double bonus;
  }
  ```
Manager Methods and Fields
- methods `setSalary`, `getName` (inherited from `Employee`)
- method `getSalary` (overridden in `Manager`)
- method `setBonus` (defined in `Manager`)
- fields `name` and `salary` (defined in `Employee`)
- field `bonus` (defined in `Manager`)

The Super/Sub Terminology
- Why is `Manager` a subclass?
- Isn’t a `Manager` superior?
- Doesn’t a `Manager` object have more fields?
- The set of managers is a subset of the set of employees
### Inheritance Hierarchies

- Real world: Hierarchies describe general/specific relationships
  - General concept at root of tree
  - More specific concepts are children
- Programming: Inheritance hierarchy
  - General superclass at root of tree
  - More specific subclasses are children

### The Substitution Principle

- Formulated by Barbara Liskov
- You can use a subclass object whenever a superclass object is expected
- Example:
  
  ```java
  Employee e;
  ...
  System.out.println("salary=" + e.getSalary());
  ```
- Can set e to Manager reference
- Polymorphism: Correct `getSalary` method is invoked

### Invoking Superclass Methods

- Can't access private fields of superclass
  ```java
  public class Manager extends Employee {
    public double getSalary() {
      return salary + bonus;
    // ERROR - salary is a private field of Employee
    }
  }
  ```
- Be careful when calling superclass method
  ```java
  public double getSalary() {
    return getSalary() + bonus;
    // ERROR - need to call super.getSalary() or otherwise this is a recursive call
  }
  ```
**Invoking Superclass Methods**
- Use super keyword
  ```java
  public double getSalary()
  {
      return super.getSalary() + bonus;
  }
  ```
- super is not a reference
- super turns off polymorphic call mechanism

**Invoking Superclass Constructors**
- Use super keyword in subclass constructor:
  ```java
  public Manager(String aName)
  {
      super(aName);
      bonus = 0;
  }
  ```
- Call to super must be first statement in subclass constructor
- If subclass constructor doesn’t call super, superclass must have constructor without parameters

**Preconditions**
- Precondition of redefined method *at most as strong*
  ```java
  public class Employee
  {
  /**
     Sets the employee salary to a given value.
     @param aSalary the new salary
     @precondition aSalary > 0
  */
  public void setSalary(double aSalary) { ... }
  }
  ```
- Can we redefine Manager.setSalary with precondition
  ```java
  salary > 100000?
  ```
- No--Could be defeated:
  ```java
  Manager m = new Manager();
  Employee e = m;
  e.setSalary(50000);
  ```

**Postconditions, Visibility, Exceptions**
- Postcondition of redefined method *at least as strong*
- Example: Employee.setSalary promises not to decrease salary
- Then Manager.setSalary must fulfill postcondition
- Redefined method cannot be more private.
  (Common error: omit public when redefining)
- Redefined method cannot throw more checked exceptions
Graphic Programming with Inheritance

- Chapter 4: Create drawings by by implementing Icon interface type
- Now: Form subclass of JPanel

```java
public class MyPanel extends JPanel
{
    public void paintComponent(Graphics g)
    {
        drawing instructions go here
    }
    ...
}
```

- Advantage: Inherit behavior from JPanel
- Example: Can attach mouse listener to JPanel

Overriding paintComponent

- Draw a car:
  ```java
  public class MyPanel extends JPanel
  {
      public void paintComponent(Graphics g)
      {
          Graphics2D g2 = (Graphics2D)g;
          car.draw(g2);
      }
      ...
  }
  ```

- Problem: Screen is corrupted when moving frame
- Remedy: call `super.paintComponent(g)`

Mouse Listeners

- Attach mouse listener to component
- Can listen to mouse events (clicks) or mouse motion events

```java
public interface MouseListener
{
    void mouseClicked(MouseEvent event);
    void mousePressed(MouseEvent event);
    void mouseReleased(MouseEvent event);
    void mouseEntered(MouseEvent event);
    void mouseExited(MouseEvent event);
}
```

```java
public interface MouseMotionListener
{
    void mouseMoved(MouseEvent event);
    void mouseDragged(MouseEvent event);
}
```
Mouse Adapters

- What if you just want to listen to mousePressed?
- Extend MouseAdapter

```java
public class MouseAdapter implements MouseListener {
    public void mouseClicked(MouseEvent event) {}
    public void mousePressed(MouseEvent event) {}
    public void mouseReleased(MouseEvent event) {}
    public void mouseEntered(MouseEvent event) {}
    public void mouseExited(MouseEvent event) {}
}
```

- Component constructor adds listener:

```java
Component constructor adds listener:
addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent event) {
        mouse action goes here
    }
});
```

Car Mover Program

- Use the mouse to drag a car shape
- Car panel has mouse + mouse motion listeners
- mousePressed remembers point of mouse press
- mouseDragged translates car shape

```java
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;
import java.util.*;

/**
 * A panel that shows a scene composed of items.
 */
public class CarPanel extends JPanel {
    public CarPanel() {
        car = new CarShape(20, 20, 50);
        addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent event) {
                mousePoint = event.getPoint();
            }
        });
        addMouseMotionListener(new MouseMotionAdapter() {
            public void mouseDragged(MouseEvent event) {
                if (mousePoint == null) return;
                Point lastMousePoint = mousePoint;
                mousePoint = event.getPoint();
                car.translate(mousePoint.getX() - lastMousePoint.getX(),
                              mousePoint.getY() - lastMousePoint.getY());
                repaint();
            }
        });
    }
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D) g;
        car.draw(g2);
        // Draw the car shape.
    }
    private CarShape car;
    private Point mousePoint;
}
```
import java.awt.*;
import java.awt.geom.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * A car shape.
 */
public class CarShape {
    
    private double x;    
    private double y;    
    private double width;

    public CarShape(int x, int y, int width) {
        this.x = x;
        this.y = y;
        this.width = width;
    }

    public boolean contains(Point2D p) {
        return x <= p.getX() && p.getY() <= x + width
            && y <= p.getY() && p.getY() <= y + width / 2;
    }

    public void translate(double dx, double dy) {
        x += dx;
        y += dy;
    }

    private int x;
    private int y;
    private int width;
}
Scene Editor

- Draws various shapes
- User can add, delete, move shapes
- User selects shape with mouse
- Selected shape is highlighted (filled in)

The SceneShape Interface Type

- keep track of selection state
- draw plain or selected shape
- move shape
- hit testing: is a point (e.g. mouse position) inside?

The SceneShape Interface Type

<table>
<thead>
<tr>
<th>SceneShape</th>
</tr>
</thead>
<tbody>
<tr>
<td>manage selection state</td>
</tr>
<tr>
<td>draw the shape</td>
</tr>
<tr>
<td>move the shape</td>
</tr>
<tr>
<td>containment testing</td>
</tr>
</tbody>
</table>
The SceneShape Interface Type

```java
public interface SceneShape {
    void setSelected(boolean b);
    boolean isSelected();
    void draw(Graphics2D g2);
    void drawSelection(Graphics2D g2);
    void translate(double dx, double dy);
    boolean contains(Point2D aPoint);
}
```

CarShape and HouseShape Classes

```java
public class CarShape implements SceneShape {
    public void setSelected(boolean b) { selected = b; }
    public boolean isSelected() { return selected; }
    private boolean selected;
}

public class HouseShape implements SceneShape {
    public void setSelected(boolean b) { selected = b; }
    public boolean isSelected() { return selected; }
    private boolean selected;
}
```

Abstract Classes

- Factor out common behavior
  (setSelected, isSelected)
- Subclasses inherit common behavior
- Some methods still undefined
  (draw, drawSelection, translate, contains)

```java
public class SelectableShape implements Item {
    public void setSelected(boolean b) { selected = b; }
    public boolean isSelected() { return selected; }
    private boolean selected;
}
```
Abstract Classes

- SelectableShape doesn’t define all SceneShape methods
- It’s abstract
- public abstract class SelectableShape implements SceneShape
- HouseShape and CarShape are concrete
- Can’t instantiate abstract class:
  SelectableShape s = new SelectableShape(); // NO
- Ok to have variables of abstract class type:
  SelectableShape s = new HouseShape(); // OK

Scene Editor

- Mouse listener selects/unselects item
- Mouse motion listener drags item
- Remove button removes selected items
- Ch6/scene1/ScenePanel.java
- Ch6/scene1/SceneEditor.java
- Ch6/scene1/HouseShape.java

```java
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;
import java.util.*;

/**
 * A panel that shows a scene composed of shapes.
 */
public class ScenePanel extends JPanel
{
    public ScenePanel()
    {
        shapes = new ArrayList();

        addMouseListener(new MouseAdapter()
        {
            public void mousePressed(MouseEvent event)
            {
                mousePoint = event.getPoint();
                for (int i = 0; i < shapes.size(); i++)
                {
                    SceneShape s = (SceneShape) shapes.get(i);
                    if (s.contains(mousePoint))
                        s.setSelected(!s.isSelected());
                }
                repaint();
            }
        });

        addMouseMotionListener(new MouseMotionAdapter()
        {
            public void mouseDragged(MouseEvent event)
            {
                Point lastMousePoint = mousePoint;
                mousePoint = event.getPoint();
                for (int i = 0; i < shapes.size(); i++)
                {
                    SceneShape s = (SceneShape) shapes.get(i);
                }
            }
        });
    }
```
if (s.isSelected())
    s.translate(mousePoint.getX() - lastMousePoint.getX(), mousePoint.getY() - lastMousePoint.getY());
    repaint();
}
}
/**
 * Adds a shape to the scene.
 * @param s the shape to add
 */
public void add(SceneShape s)
{
    shapes.add(s);
    repaint();
}
/**
 * Removes all selected shapes from the scene.
 */
public void removeSelected()
{
    for (int i = shapes.size() - 1; i >= 0; i--)
        if (s.isSelected()) shapes.remove(i);
    repaint();
}
public void paintComponent(Graphics g)
{
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D) g;
    for (int i = 0; i < shapes.size(); i++)
        Shape shape = (Shape) shapes.get(i);
        shape.paint(g2);
}

import java.awt.;
import java.awt.geom.;
import java.awt.event.;
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public static void main(String[] args)
{
    JFrame frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    Container contentPane = frame.getContentPane();
    final ScenePanel panel = new ScenePanel();
    JButton houseButton = new JButton("House");
    houseButton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            panel.add(new HouseShape(20, 20, 50));
            repaint();
        }
    });
    JButton carButton = new JButton("Car");
    carButton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            panel.add(new CarShape(20, 20, 50));
            repaint();
        }
    });
    JButton removeButton = new JButton("Remove");
    removeButton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            panel.removeSelected();
            repaint();
        }
    });
    JPanel buttons = new JPanel();
    buttons.add(houseButton);
    buttons.add(carButton);
    buttons.add(removeButton);
    contentPane.add(panel, BorderLayout.CENTER);
    JButton addButton = new JButton("Add");
    addButton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            panel.addSceneShape();
            repaint();
        }
    });
Uniform Highlighting Technique

- Old approach: each shape draws its selection state
- Inconsistent
- Better approach: shift, draw, shift, draw, restore to original position
- Define in SelectableShape

```java
public void drawSelection(Graphics2D g2) {
    translate(1, 1);
    draw(g2);
    translate(1, 1);
    draw(g2);
    translate(-2, -2);
}
```

---

Uniform Highlighting Technique

```java
import java.awt.*;
import java.awt.geom.*;

/**
   * A house shape.
   */
public class HouseShape extends SelectableShape {
    public HouseShape(int x, int y, int width) {
        this.x = x;
        this.y = y;
        this.width = width;
    }

    public void draw(Graphics2D g2) {
        Rectangle2D.Double base = new Rectangle2D.Double(x, y + width, width, width);
        // the left bottom of the roof
        Point2D.Double r1 = new Point2D.Double(x, y + width);
        // the top of the roof
        Point2D.Double r2 = new Point2D.Double(x + width / 2, y);
        // the right bottom of the roof
        Point2D.Double r3 = new Point2D.Double(x + width, y + width);
        Line2D.Double roofLeft = new Line2D.Double(r1, r2);
        Line2D.Double roofRight = new Line2D.Double(r2, r3);
        g2.draw(base);
        g2.draw(roofLeft);
        g2.draw(roofRight);
    }

    public void translate(double dx, double dy) {
        x += dx;
        y += dy;
    }
}
```
Template Method

- drawSelection calls draw
- Must declare draw in SelectableShape
- No implementation at that level!
- Declare as abstract method

```java
public abstract void draw(Graphics2D g2)
```

- DrawSelection method calls draw, translate
- drawSelection doesn’t know which methods—polymorphism
- drawSelection is a template method

---

import java.awt.*;
import java.awt.geom.*;

/**
 * A shape that manages its selection state.
 */
public abstract class SelectableShape implements SceneShape
{

    // the left bottom of the roof
    Point2D.Double roofLeft = new Point2D.Double(x, y);
    // the right bottom of the roof
    Point2D.Double roofRight = new Point2D.Double(x + width, y);
    // the top of the roof
    Point2D.Double r1 = new Point2D.Double(x, y + width / 2);
    // the left top of the roof
    Point2D.Double r2 = new Point2D.Double(x + width / 2, y);
    // the right top of the roof
    Point2D.Double r3 = new Point2D.Double(x + width, y + width / 2);
    // the base line of the roof
    Line2D.Double base = new Line2D.Double(x, y + width, x + width, y + width);
    // the left bottom of the roof
    Line2D.Double roofLeft = new Line2D.Double(x, y, x + width / 2, y + width / 2);
    // the right bottom of the roof
    Line2D.Double roofRight = new Line2D.Double(x + width / 2, y, x + width, y + width / 2);
    // the base line of the roof
    Line2D.Double roofBase = new Line2D.Double(x, y, x + width, y);
}
```
**TEMPLATE METHOD Pattern**

**Context**
1. An algorithm is applicable for multiple types.
2. The algorithm can be broken down into primitive operations. The primitive operations can be different for each type.
3. The order of the primitive operations doesn’t depend on the type.

**Solution**
1. Define a superclass that has a method for the algorithm and abstract methods for the primitive operations.
2. Implement the algorithm to call the primitive operations in the appropriate order.
3. Do not define the primitive operations in the superclass, or define them to have appropriate default behavior.
4. Each subclass defines the primitive operations but not the algorithm.

---

**Diagram:**

AbstractClass (abstract)
- templateMethod()
- primitiveOp1()
- primitiveOp2()

ConcreteClass
- primitiveOp1()
- primitiveOp2()

Calls primitiveOp1(), primitiveOp2()

name in Design Pattern | Actual Name (Selectable shapes)
---|---
AbstractClass | SelectableShape
ConcreteClass | CarShape, HouseShape
templateMethod() | drawSelection
primitiveOp1(), primitiveOp2() | translate, draw
### Compound Shapes

- **GeneralPath**: sequence of shapes
  
  ```java
generalPath = new GeneralPath();
path.append(new Rectangle(...), false);
```  

- Advantage: Containment test is free
  
  ```java
path.contains(aPoint);
```

- Ch6/scene3/CompoundShape.java
- Ch6/scene3/HouseShape.java

---

```java
import java.awt.*;
import java.awt.geom.*;

/**
 * A house shape.
 */
public abstract class HouseShape extends SelectableShape {

    @param width the width of the bounding rectangle
    @param y the top of the bounding rectangle
    @param x the left of the bounding rectangle
    Constructs a house shape.
    {
        path = new GeneralPath();
        g2.draw(path);
    }

    private GeneralPath path;
}
```
**Access to Superclass Features**

- Why does the HouseShape constructor call `add`?

```java
public HouseShape()
{
    add(new Rectangle(...));
    add(new Triangle(...));
}
```

- Why not just `path.append(new Rectangle(...));`?

HouseShape inherits `path` field
HouseShape can’t access `path`
`path` is private to superclass

**Protected Access**

- Make CompoundShape.add method `protected`
- Protects HouseShape: other classes can’t add graffiti
- Protected features can be accessed by subclass methods...
- ...and by methods in the same package
- Bad idea to make fields protected
  ```java
  protected GeneralPath path; // DON’T
  ```
- Ok to make methods protected
  ```java
  protected void add(Shape s) // GOOD
  ```
- Protected interface separate from public interface

**Hierarchy of Swing Components**

- Base of hierarchy: Component
- Huge number of common methods:
  ```java
  int getWidth();
  int getHeight();
  Dimension getPreferredSize();
  void setBackground(Color c)
  ```
- Most important subclass: Container

**Hierarchy of Swing Components**

[Diagram of the hierarchy of Swing components]

- Component
  - Container
    - JComponent
      - JButton
      - JCheckBox
      - JRadioButton
      - JMenuItem
      - JTextField
      - JTextArea
      - JScrollPane
      - AbstractButton
Hierarchy of Swing Components

- History: First came AWT, Abstract Window Toolkit
- Used native components
- Subtle platform inconsistencies
- Write once, run anywhere -> Write once, debug everywhere
- Swing paints components onto blank windows
- Supports multiple look and feel implementations

Hierarchy of Swing Components

- Base of Swing components: JComponent
- Subclass of Container
- Some Swing components are containers
- Java has no multiple inheritance
- JLabel, JPanel, ... are subclasses of JComponent
- Intermediate classes AbstractButton, JTextComponent

Look and Feel

- First version of Java: few shapes, integer coordinates
- Point
- Rectangle
- Polygon
- Java 2: sophisticated shapes, floating-point coordinates
- Point2D
- Rectangle2D
- RoundRectangle2D
- Line2D
- Ellipse2D
- Arc2D
- QuadCurve2D
- CubicCurve2D
- GeneralPath
- Area
- All but Point2D implement Shape interface type

Hierarchy of Geometrical Shapes
Hierarchy of Geometrical Shapes

Rectangular Shapes

- Subclasses of RectangularShape:
  - Rectangle2D
  - RoundRectangle2D
  - Ellipse2D
  - Arc2D

- RectangularShape has useful methods
  - getCenterX/getCenterY
  - getMinX/getMinY
  - getMaxX/getMaxY
  - getWidth/getHeight
  - setFrameFromCenter/setFrameFromDiagonal

Float/Double Classes

- Each class has two subclasses, e.g.
  - Rectangle2D.Double
  - Rectangle2D.Float

- Are also inner classes!
  - (Just to avoid even longer class names)
- Implementations have double/float fields
- Most methods have double parameters/return values
**Float/Double Classes**

```java
public class Rectangle2D {
    public static class Float extends Rectangle2D {
        ... public float x; 
        public float y; 
        public float width; 
        public float height; 
    }
    ...}
```

**Float/Double Classes**

```java
public static class Double extends Rectangle2D {
    public double getX() { return x; }
    public double getY() { return y; }
    public double getWidth() { return width; }
    public double getHeight() { return height; }
    public void setRect(double x, double y, double w, double h) {
        this.x = x; this.y = y;
        this.width = w; this.height = h;
    }
    public void setRect(double x, double y, double w, double h) {
        this.x = (float)x; this.y = (float)y;
        this.width = (float)w; this.height = (float)h;
    }
    ...
    public double x;
    public double y;
    public double width;
    public double height;
}
```

**Float/Double Classes**

```java
Rectangle2D class has no instance variables
Template Method Pattern at work:
public boolean contains(double x, double y) {
    double x0 = getX();
    double y0 = getY();
    return x >= x0 && y >= y0 &&
    x < x0 + getWidth() &&
    y < y0 + getHeight();
}
```

**Float/Double Classes**

- Rectangle2D class has no instance variables
- Template Method Pattern at work:
  ```java
  public boolean contains(double x, double y) {
      double x0 = getX();
      double y0 = getY();
      return x >= x0 && y >= y0 &&
      x < x0 + getWidth() &&
      y < y0 + getHeight();
  }
  ```

- No need to use inner class after construction
  ```java
  Rectangle2D rect = new Rectangle2D.Double(5, 10, 20, 30);
  ```

**TEMPLATE METHOD Pattern**

<table>
<thead>
<tr>
<th>Name in Design Pattern</th>
<th>Actual Name (Rectangles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbstractClass</td>
<td>Rectangle</td>
</tr>
<tr>
<td>ConcreteClass</td>
<td>Rectangle2D.Double</td>
</tr>
<tr>
<td>templateMethod()</td>
<td>contains</td>
</tr>
<tr>
<td>primitiveOpn()</td>
<td>getX, getY, getWidth, getHeight</td>
</tr>
</tbody>
</table>

- No need to use inner class after construction

  ```java
  Rectangle2D rect = new Rectangle2D.Double(5, 10, 20, 30);
  ```
Hierarchy of Exception Classes

- Base of hierarchy: Throwable
- Two subclasses: Error, Exception
- Subclasses of Error: fatal
  (out of memory, assertion failure)
- Subclasses of Exception:
  - Lots of checked exceptions
    (I/O, class not found)
  - RuntimeException--its subclasses are unchecked
    (null pointer, index out of bounds)

Defining Exception Classes

- Decide exception should be checked
- Subclass Exception or RuntimeException
- Provide two constructors

```java
public class IllegalFormatException extends Exception
{
    public IllegalFormatException()
    {
    }
    public IllegalFormatException(String reason)
    {
        super(reason);
    }
}
```

- Throw exception when needed:
  ```java
  throw new IllegalFormatException("number expected");
  ```

Catching Exceptions

- Can have multiple catch clauses:
  ```java
  try
  {
  code that may throw exceptions
  } catch (ExceptionType1 exception1)
  { handler for ExceptionType1 }
  catch (ExceptionType2 exception2)
  { handler for ExceptionType2 }
  ...
  ```
- Can catch by superclass:
  ```java
  catch (IOException exception)
  catches FileNotFoundException
  ```
When Not to Use Inheritance

- From a tutorial for a C++ compiler:

```java
public class Point {
   public Point(int anX, int aY) { ... }
   public void translate(int dx, int dy) { ... }
   private int x;
   private int y;
}
```

```java
public class Circle extends Point // DON'T {
   public Circle(Point center, int radius) { ... }
   public void draw(Graphics g) { ... }
   private int radius;
}
```

- Huh? A circle isn’t a point.
- By accident, inherited `translate` works for circles
- Same tutorial makes Rectangle a subclass of Point:

```java
public class Rectangle extends Point // DON'T {
   public Rectangle(Point corner1, Point corner2) { ... }
   public void draw(Graphics g) { ... }
   public void translate(int dx, int dy) { ... }
   private Point other;
}
```

- That’s even weirder:

```java
public void translate(double dx, double dy) {
   super.translate(dx, dy);
   other.translate(dx, dy);
}
```

- Why did they do that?
- Wanted to avoid abstract class `Shape`
- Remedy: Use aggregation.

- Circle, Rectangle classes have points

When Not to Use Inheritance

- Java standard library:

```java
public class Stack extends Vector // DON'T {
   Object pop() { ... }
   void push(Object item) { ... }
   ... 
}
```

- Bad idea: Inherit all `Vector` methods
- Can insert/remove in the middle of the stack
- Remedy: Use aggregation

```java
public class Stack {
   ... 
   private Vector elements;
}
```