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)

next .... [Slide 1] ....
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
Modeling Specialization

- Manager class adds new method: setBonus
- Manager class overrides existing method: getSalary
- Adds salary and bonus
- public class Manager extends Employee
  {
  public Manager(String aName) { ... }
  public void setBonus(double aBonus)
  { bonus = aBonus; } // new method
  public double getSalary()
  { ... } // overrides Employee method
  private double bonus; // new field
  }
Modeling Specialization

```
Employee
  name
  salary
  getName()
  getSalary()
  setSalary()

Manager
  bonus
  setBonus()
  getSalary()

Overrides Employee method
```

previous | start | next .... [Slide 5] ....
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
The Super/Sub Terminology

Managers

Employees

previous | start | next .... [Slide 8] ....
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
Inheritance Hierarchies

- Object
  - Employee
    - Manager
      - Executive
    - Clerical Staff Member
      - Receptionist
      - Secretary
    - Technical Staff Member
      - Software Developer
      - Test Engineer
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 
  // else 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);
      // calls superclass constructor
      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 `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 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)`
Overriding `paintComponent`
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);
}

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
addMouseListener(new MouseAdapter () {
    public void mousePressed(MouseEvent event) {
        mouse action goes here
    }
});
```

previous | start | next .... [Slide 21] ....
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
- Ch6/car/CarPanel.java
- Ch6/car/CarMover.java
- Ch6/car/CarShape.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();
                if (!car.contains(mousePoint))
                    mousePoint = null;
            }
        });
        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();
            }
        });
    }
}
```java
42: public void paintComponent(Graphics g) {
43:     super.paintComponent(g);
44:     Graphics2D g2 = (Graphics2D) g;
45:     car.draw(g2);
46: }
47: }
48: }
49: }
50: private CarShape car;
51: private Point mousePoint;
52: }
```
import java.awt.*;
import java.awt.geom.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * A program that allows users to move a car with the mouse.
 */
public class CarMover {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container contentPane = frame.getContentPane();
        contentPane.add(new CarPanel());
        frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
        frame.show();
    }

    private static final int FRAME_WIDTH = 400;
    private static final int FRAME_HEIGHT = 400;
}
```java
import java.awt.*;
import java.awt.geom.*;

/**
 * A car shape.
 */
public class CarShape
{
    /**
     * Constructs a car shape.
     * @param x the left of the bounding rectangle
     * @param y the top of the bounding rectangle
     * @param width the width of the bounding rectangle
     */
    public CarShape(int x, int y, int width)
    {
        this.x = x;
        this.y = y;
        this.width = width;
    }

    public void draw(Graphics2D g2)
    {
        Rectangle2D.Double body
        = new Rectangle2D.Double(x, y + width / 6,
                                 width - 1, width / 6);
        Ellipse2D.Double frontTire
        = new Ellipse2D.Double(x + width / 6, y + width / 3,
                               width / 6, width / 6);
        Ellipse2D.Double rearTire
        = new Ellipse2D.Double(x + width * 2 / 3,
                                y + width / 3,
                                width / 6, width / 6);

        // the bottom of the front windshield
        Point2D.Double r1
        = new Point2D.Double(x + width / 6, y + width / 6);
        // the front of the roof
        Point2D.Double r2
        = new Point2D.Double(x + width / 3, y);
        // the rear of the roof
```
Point2D.Double r3 = new Point2D.Double(x + width * 2 / 3, y);
// the bottom of the rear windshield
Point2D.Double r4 = new Point2D.Double(x + width * 5 / 6, y + width / 6);
Line2D.Double frontWindshield = new Line2D.Double(r1, r2);
Line2D.Double roofTop = new Line2D.Double(r2, r3);
Line2D.Double rearWindshield = new Line2D.Double(r3, r4);
g2.draw(body);
g2.draw(frontTire);
g2.draw(rearTire);
g2.draw(frontWindshield);
g2.draw(roofTop);
g2.draw(rearWindshield);

private int x;
private int y;
private int width;

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

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

Car Mover Program
Scene Editor

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

previous | start | next .... [Slide 25] ....
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 {
    private boolean selected;
    public void setSelected(boolean b) { selected = b; }
    public boolean isSelected() { return selected; }
}

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

```
«interface»
Scene
Shape

isSelected()
setSelected()
draw()
drawSelection()
translate()
contains()

Selectable
Shape
{abstract}

selected
isSelected()
setSelected()

Car
Shape

draw()
drawSelection()
translate()
contains()

House
Shape

draw()
drawSelection()
translate()
contains()
```
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:

  ```java
  SelectableShape s = new SelectableShape(); // NO
  ```

- Ok to have variables of abstract class type:

  ```java
  SelectableShape s = new HouseShape(); // OK
  ```
Abstract Classes and Interface Types

- Abstract classes can have fields
- Interface types can only have constants (public static final)
- Abstract classes can define methods
- Interface types can only declare methods
- A class can implement any number of interface types
- In Java, a class can extend only one other class
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
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.contains(mousePoint))
                        s.setSelected(!s.isSelected());
                }
                repaint();
            }
        });
    }
}

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--)
    {
        SceneShape s = (SceneShape) shapes.get(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++)
    {
        SceneShape s = (SceneShape) shapes.get(i);
        s.draw(g2);
if (s.isSelected())
    s.drawSelection(g2);
}

private ArrayList shapes;
private Point mousePoint;
import java.awt.*;
import java.awt.geom.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * A program that allows users to edit a scene composed of items.
 */
public class SceneEditor
{
    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));
            }
        });

        JButton carButton = new JButton("Car");
        carButton.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                panel.add(new CarShape(20, 20, 50));
            }
        });

        JButton removeButton = new JButton("Remove");
        removeButton.addActionListener(new
ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        panel.removeSelected();
    }
}

JPanel buttons = new JPanel();
buttons.add(houseButton);
buttons.add(carButton);
buttons.add(removeButton);

contentPane.add(panel, BorderLayout.CENTER);
contentPane.add(buttons, BorderLayout.NORTH);
frame.setSize(300, 300);
frame.show();
import java.awt.*;
import java.awt.geom.*;

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

    /**
     * Constructs a house shape.
     * @param x the left of the bounding rectangle
     * @param y the top of the bounding rectangle
     * @param width the width of the bounding rectangle
     */
    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);
    }
}
public void drawSelection(Graphics2D g2) {
    Rectangle2D.Double base = new Rectangle2D.Double(x, y + width, width, width);
    g2.fill(base);
}

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

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

private int x;
private int y;
private int width;
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
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)
  ```
- Defined in `CarShape`, `HouseShape`
- `drawSelection` method calls `draw`, `translate`
- `drawSelection` doesn’t know which methods--polymorphism
- `drawSelection` is a `template method`
- `Ch6/scene2/SelectableShape.java`
- `Ch6/scene2/HouseShape.java`
import java.awt.*;
import java.awt.geom.*;

/**
 * A shape that manages its selection state.
 */
public abstract class SelectableShape implements SceneShape {
    public void setSelected(boolean b) {
        selected = b;
    }
    public boolean isSelected() {
        return selected;
    }
    public void drawSelection(Graphics2D g2) {
        translate(1, 1);
        draw(g2);
        translate(1, 1);
        draw(g2);
        translate(-2, -2);
    }
    private boolean selected;
}
import java.awt.*;
import java.awt.geom.*;

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

    /**
     * Constructs a house shape.
     * @param x the left of the bounding rectangle
     * @param y the top of the bounding rectangle
     * @param width the width of the bounding rectangle
     */
    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);
    }
}
42:       g2.draw(base);
43:       g2.draw(roofLeft);
44:       g2.draw(roofRight);
45:   }
46:
47:   public boolean contains(Point2D p)
48:   {
49:       return x <= p.getX() && p.getX() <= x + width
50:           && y <= p.getY() && p.getY() <= y + 2 * width;
51:   }
52:
53:   public void translate(double dx, double dy)
54:   {
55:       x += dx;
56:       y += dy;
57:   }
58:
59:   private int x;
60:   private int y;
61:   private int width;
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.
TEMPLATE METHOD Pattern

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.
TEMPLATE METHOD Pattern

AbstractClass
{abstract}

- templateMethod()
- primitiveOp1()
- primitiveOp2()

Calls primitiveOp1(), primitiveOp2()

Abstract methods

ConcreteClass

- primitiveOp1()
- primitiveOp2()
# TEMPLATE METHOD Pattern

<table>
<thead>
<tr>
<th>Name in Design Pattern</th>
<th>Actual Name (Selectable shapes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbstractClass</td>
<td>SelectableShape</td>
</tr>
<tr>
<td>ConcreteClass</td>
<td>CarShape, HouseShape</td>
</tr>
<tr>
<td>templateMethod()</td>
<td>drawSelection</td>
</tr>
<tr>
<td>primitiveOp1(), primitiveOp2()</td>
<td>translate, draw</td>
</tr>
</tbody>
</table>

previous | start | next .... [Slide 41] ....
Compound Shapes

- GeneralPath: sequence of shapes

  GeneralPath path = new GeneralPath();
  path.append(new Rectangle(...), false);
  path.append(new Triangle(...), false);
  g2.draw(path);

- Advantage: Containment test is free
  path.contains(aPoint);
- Ch6/scene3/CompoundShape.java
- Ch6/scene3/HouseShape.java
import java.awt.*;
import java.awt.geom.*;

/**
 * A scene shape that is composed of multiple geometric shapes.
 */
public abstract class CompoundShape extends SelectableShape {

    public CompoundShape() {
        path = new GeneralPath();
    }

    protected void add(Shape s) {
        path.append(s, false);
    }

    public boolean contains(Point2D aPoint) {
        return path.contains(aPoint);
    }

    public void translate(double dx, double dy) {
        AffineTransform t = AffineTransform.getTranslateInstance(dx, dy);
        path.transform(t);
    }

    public void draw(Graphics2D g2) {
        g2.draw(path);
    }

    private GeneralPath path;
}
import java.awt.*;
import java.awt.geom.*;

/**
 * A house shape.
 */
public class HouseShape extends CompoundShape {
    /**
     * Constructs a house shape.
     * @param x the left of the bounding rectangle
     * @param y the top of the bounding rectangle
     * @param width the width of the bounding rectangle
     */
    public HouseShape(int x, int y, int width) {
        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);
        add(base);
        add(roofLeft);
        add(roofRight);
    }
}
Compound Shapes

```
«interface»
Scene
Shape

Selectable
Shape {abstract}

setSelected()
isSelected()
drawSelection()

Compound
Shape {abstract}

draw()
translate()
contains()

House
Shape
```
Access to Superclass Features

- Why does the HouseShape constructor call add?

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

- Why not just
  ```java
  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
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
Look and Feel
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
Hierarchy of Geometrical Shapes

- 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
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

Rectangle2D
{abstract}

Rectangle2D
.Double

Rectangle2D
.Float

previous | start | next .... [Slide 55] ....
Float/Double Classes

public class Rectangle2D
{
    public static class Float 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 = (float)x; this.y = (float)y;
            this.width = (float)w; this.height = (float)h;
        }
        public void setRect(float x, float y, float w, float h)
        {
            this.x = x; this.y = y;
            this.width = w; this.height = h;
        }
        ...
        public float x;
        public float y;
        public float width;
        public float height;
    }
    ...
}

previous | start | next .... [Slide 56] ....
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 double x;
    public double y;
    public double width;
    public double height;
}...
```
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>

previous | start | next .... [Slide 59] ....
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)
Hierarchy of Exception Classes

- Throwable
  - Exception
    - RuntimeException
      - NullPointerException
      - IndexOutOfBoundsException
    - IOException
    - ClassNotFoundException
      - FileNotFoundException
  - Error
Catching Exceptions

- Can have multiple `catch` clauses:

```java
try {
    code that may throw exceptions
} catch (ExceptionType1 exception1) {
    handler for ExceptionType1
} catch (ExceptionType2 exception1) {
    handler for ExceptionType2
} . . .
```

- Can catch by superclass:
  - `catch (IOException exception)` catches `FileNotFoundException`
  - `catch (IOException exception)` catches `FileNotFoundException`
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");
  ```
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;
}

public class Circle extends Point // DON’T
{
    public Circle(Point center, int radius) { ... }
    public void draw(Graphics g) { ... }
    private int radius;
}
```
When Not to Use Inheritance

- 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;
}
```
When Not to Use Inheritance

- 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;
  }
  ```