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 Geometric 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

```java
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
Modeling Specialization

• 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
Inheritance Hierarchies

• A Hierarchy of Employee Classes:
The Substitution Principle

• Formulated by Barbara Liskov

• You can use a subclass object whenever a superclass object is expected

• Example:

  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--private field
    }
    ...
  }
  ```

- Be careful when calling superclass method
  ```java
  public double getSalary() {
    return getSalary() + bonus; // ERROR--recursive call
  }
  ```
Invoking Superclass Methods

• Use super keyword

    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
  
  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:
  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 JComponent

    public class MyComponent extends JComponent
    {
        public void paintComponent(Graphics g)
        {
            drawing instructions go here
        }
        ...
    }

• Advantage: Inherit behavior from JComponent

• Example: Can attach mouse listener to JComponent
We will develop a program that allows a user to move a car by dragging it with the mouse.
In the following sections, that program will be enhanced to show a scene composed of arbitrary shapes.
The CarComponent class stores a reference to a CarShape object.
Its paintComponent method draws the shape:

```java
public class CarComponent extends JComponent {
    private CarShape car;

    public void paintComponent(Graphics g) {
        Graphics2D g2 = (Graphics2D) g;
        car.draw(g2);
    }
}
```
Listener Interface Types and Adapter Classes

- 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 {
    // consumes more CPU cycles than MouseListener
    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
    }
  });
  ```
The Car Mover Program

01: import java.awt.*;
02: import java.awt.geom.*;
03:
04: /**
05:    A car shape.
06: */
07: public class CarShape
08: {
09:    /**
10:       Constructs a car shape.
11:       @param x the left of the bounding rectangle
12:       @param y the top of the bounding rectangle
13:       @param width the width of the bounding rectangle
14:    */
15:    public CarShape(int x, int y, int width)
16:    { 
17:        this.x = x;
18:        this.y = y;
19:        this.width = width;
20:    }
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 Car Mover Program

```java
35:     // The bottom of the front windshield
36:     Point2D.Double r1
37:         = new Point2D.Double(x + width / 6, y + width / 6);
38:     // The front of the roof
39:     Point2D.Double r2
40:         = new Point2D.Double(x + width / 3, y);
41:     // The rear of the roof
42:     Point2D.Double r3
43:         = new Point2D.Double(x + width * 2 / 3, y);
44:     // The bottom of the rear windshield
45:     Point2D.Double r4
46:         = new Point2D.Double(x + width * 5 / 6, y + width / 6);
47:     Line2D.Double frontWindshield
48:         = new Line2D.Double(r1, r2);
49:     Line2D.Double roofTop
50:         = new Line2D.Double(r2, r3);
51:     Line2D.Double rearWindshield
52:         = new Line2D.Double(r3, r4);
```
The Car Mover Program

54:     g2.draw(body);
55:     g2.draw(frontTire);
56:     g2.draw(rearTire);
57:     g2.draw(frontWindshield);
58:     g2.draw(roofTop);
59:     g2.draw(rearWindshield);
60: }

61:
62: public boolean contains(Point2D p)
63: {
64:     return x <= p.getX() && p.getX() <= x + width
65:          && y <= p.getY() && p.getY() <= y + width / 2;
66: }

67:
68: public void translate(int dx, int dy)
69: {
70:     x += dx;
71:     y += dy;
72: }

73:
74: private int x;
75: private int y;
76: private int width;
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;
import java.util.*;

public class CarComponent extends JComponent {
    public CarComponent() {
        car = new CarShape(20, 20, 50);
        addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent event) {
                mousePoint = event.getPoint();
                if (!car.contains(mousePoint))
                    mousePoint = null;
            }
        });
    }
}
The Car Mover Program

26:   addMouseMotionListener(new
27:     MouseMotionAdapter()
28:     {
29:        public void mouseDragged(MouseEvent event)
30:           {
31:              if (mousePoint == null) return;
32:              Point lastMousePoint = mousePoint;
33:              mousePoint = event.getPoint();
34:              double dx = mousePoint.getX() - lastMousePoint.getX();
35:              double dy = mousePoint.getY() - lastMousePoint.getY();
36:              car.translate((int) dx, (int) dy);
37:              repaint();
38:           }
39:     });
40:   }
public void paintComponent(Graphics g) {
    Graphics2D g2 = (Graphics2D) g;
    car.draw(g2);
}

private CarShape car;
private Point mousePoint;
The Car Mover Program

01: import java.awt.*;
02: import java.awt.geom.*;
03: import java.awt.event.*;
04: import javax.swing.*;
05:
06: /**
07:   A program that allows users to move a car with the mouse.
08: */
09: public class CarMover
10: {
11:     public static void main(String[] args)
12:     {
13:         JFrame frame = new JFrame();
14:         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15:         frame.add(new CarComponent());
16:         frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
17:         frame.setVisible(true);
18:     }
19: }
20:
21: private static final int FRAME_WIDTH = 400;
22: private static final int FRAME_HEIGHT = 400;
23: }
The 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

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?

<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>
Scene Editor

• The SceneShape Interface Type:

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

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;
}
```
The Car Mover Program

• We can improve the design by moving the commonality of CarShape and HouseShape to a new superclass called SelectableShape.

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

  public class CarShape extends SelectableShape { . . . }
  public class HouseShape extends SelectableShape { . . . }

• We have a problem with the SelectableShape class. It does not define all of the methods of the SceneShape interface type. Four methods are left undefined:

  void draw(Graphics2D g2)
  void drawSelection(Graphics2D g2)
  void translate (double dx, double dy)
  boolean contains(Point2D appoint)

• These methods need to be defined in real subclasses.
The Car Mover Program
Abstract Classes

• Factor out common behavior
  (setSelected, isSelected)
• Subclasses inherit common behavior
• Some methods still undefined
  (draw, drawSelection, translate, contains)
• We cannot create instances of SelectableShape because it has abstract (undefined) methods.
• We note that by the keyword abstract:
  public abstract class SelectableShape implements SceneShape {
  . . . }
• HouseShape and CarShape are concrete subclasses that define the remaining methods.
• We can create instances of the concrete subclasses:
  SelectableShape shape = new HouseShape();
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:
  - SelectableShape s = new SelectableShape(); // NO
- Ok to have variables of abstract class type:
  - 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
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;
import java.util.*;

public class SceneComponent extends JComponent {

    public SceneComponent() {
        shapes = new ArrayList<SceneShape>();
    }

    addMouseListener(new MouseAdapter() {
        public void mousePressed(MouseEvent event)
        {
            mousePoint = event.getPoint();
            for (SceneShape s : shapes)
            {
                if (s.contains(mousePoint))
                    s.setSelected(!s.isSelected());
            }
            repaint();
        }
    });
}
addMouseMotionListener(new MouseMotionAdapter() {
    public void mouseDragged(MouseEvent event) {
        Point lastMousePoint = mousePoint;
        mousePoint = event.getPoint();
        for (SceneShape s : shapes) {
            if (s.isSelected()) {
                double dx = mousePoint.getX() - lastMousePoint.getX();
                double dy = mousePoint.getY() - lastMousePoint.getY();
                s.translate((int) dx, (int) dy);
            }
        }
        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 = shapes.get(i);
        if (s.isSelected()) shapes.remove(i);
    }
    repaint();
}
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D) g;
    for (SceneShape s : shapes) {
        s.draw(g2);
        if (s.isSelected())
            s.drawSelection(g2);
    }
}

private ArrayList<SceneShape> 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);
        final SceneComponent scene = new SceneComponent();
        JButton houseButton = new JButton("House");
        houseButton.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                scene.add(new HouseShape(20, 20, 50));
            }
        });
Scene Editor
29: JButton carButton = new JButton("Car");
30: carButton.addActionListener(new
31:     ActionListener()
32:     {
33:         public void actionPerformed(ActionEvent event)
34:         {
35:             scene.add(new CarShape(20, 20, 50));
36:         }
37:     });
38: JButton removeButton = new JButton("Remove");
39: removeButton.addActionListener(new
40:     ActionListener()
41:     {
42:         public void actionPerformed(ActionEvent event)
43:         {
44:             scene.removeSelected();
45:         }
46:     });
Scene Editor

49: JPanel buttons = new JPanel();
50: buttons.add(houseButton);
51: buttons.add(carButton);
52: buttons.add(removeButton);
53: 
54: frame.add(scene, BorderLayout.CENTER);
55: frame.add(buttons, BorderLayout.NORTH);
56: frame.setSize(300, 300);
57: frame.setVisible(true);
58: }
59: }
Refactoring

• Martin Fowler coined the term “refactoring” for restructuring code in a disciplined way.
• There are a large number of refactoring rules.
• Each rule starts with a brief explanation of the possible benefits of applying the restructuring and then contains before and after scenarios.
• As opposed to Design Patterns, Refactoring Rules tell you how to improve code that is already written.
Refactoring

- **Extract Superclass:**
  - **Symptom:** You have two classes with similar features
  - **Remedy:** Create a superclass and move the common features to the superclass.
Refactoring

• Introduce Explaining Variable
  • Symptom: You have an expression that is hard to understand
  • Remedy: Put the value of the expression in a temporary variable whose name explains the purpose of the expression.

```java
int xdistance = mousePoint.getx() - lastMousePoint.getx();
int ydistance = mousePoint.gety() - lastMousePoint.gety();
car.translate(xdistance, ydistance);
```
The Car Mover Program

```java
01: import java.awt.*;
02: import java.awt.geom.*;
03:
04: /**
05:    A house shape.
06: */
07: public class HouseShape extends SelectableShape
08: {
09:    /**
10:       Constructs a house shape.
11:       @param x the left of the bounding rectangle
12:       @param y the top of the bounding rectangle
13:       @param width the width of the bounding rectangle
14:    */
15:    public HouseShape(int x, int y, int width)
16:    {
17:        this.x = x;
18:        this.y = y;
19:        this.width = width;
20:    }
21:
```
```java
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 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(int dx, int 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);
    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
  
  ```
  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
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;
}
```java
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);

    g2.draw(base);
    g2.draw(roofLeft);
    g2.draw(roofRight);
}
public boolean contains(Point2D p) {
    return x <= p.getX() && p.getX() <= x + width
        && y <= p.getY() && p.getY() <= y + 2 * width;
}

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

private int x;
private int y;
private int width;
TEMPLATE METHOD Pattern

• Context

• An algorithm is applicable for multiple types.
• The algorithm can be broken down into primitive operations. The primitive operations can be different for each type
• The order of the primitive operations doesn't depend on the type
• TEMPLATE METHOD Pattern

• Solution
• Define a superclass that has a method for the algorithm and abstract methods for the primitive operations.
• Implement the algorithm to call the primitive operations in the appropriate order.
• Do not define the primitive operations in the superclass, or define them to have appropriate default behavior.
• Each subclass defines the primitive operations but not the algorithm.
Template Method Pattern

<table>
<thead>
<tr>
<th>Name in Design Pattern</th>
<th>Actual Name</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>
Protected Interfaces

- **Compound Shapes:**
  - GeneralPath: sequence of shapes

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

- Advantage: Containment test is free
  
```java
path.contains(aPoint);
```
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(int dx, int dy) {
    path.transform(AffineTransform.getTranslateInstance(dx, dy));
}

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

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

public class HouseShape extends CompoundShape {

    public HouseShape(int x, int y, int width) {
        Rectangle2D.Double base = new Rectangle2D.Double(x, y + width, width, width);
    }
}
Compound Shapes

20: // The left bottom of the roof
21: Point2D.Double r1 = new Point2D.Double(x, y + width);
22: // The top of the roof
23: Point2D.Double r2 = new Point2D.Double(x + width / 2, y);
24: // The right bottom of the roof
25: Point2D.Double r3 = new Point2D.Double(x + width, y + width);
26: Line2D.Double roofLeft = new Line2D.Double(r1, r2);
27: Line2D.Double roofRight = new Line2D.Double(r2, r3);
28: add(base);
29: add(roofLeft);
30: add(roofRight);
31: }
32: }
33: }
34: add(base);
35: add(roofLeft);
36: add(roofRight);
37: }
38: }
39: }
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
• protected GeneralPath path;  // DON'T
• Ok to make methods protected
• 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:
  • int getWidth()
  • int getHeight()
  • Dimension getPreferredSize()
  • void setBackground(Color c)
• . . .
• Most important subclass: Container
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

- 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, JButton, ... 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**
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
  ```
Float/Double Classes

```
Rectangle2D
{abstract}

Rectangle2D
.Double

Rectangle2D
.Float
```
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(float x, float y, float w, float 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 float x;
        public float y;
        public float width;
        public float height;
    }
...
Float/Double Classes

• ...
• 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:

  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

  Rectangle2D rect = new Rectangle2D.Double(5, 10, 20, 30);
<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>
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
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
Defining Exception Classes

• Decide exception should be checked
• Subclass Exception or RuntimeException
• Provide two constructors
• public class IllegalFormatException extends Exception
  {
    public IllegalFormatException() {}
    public IllegalFormatException(String reason)
    {
      super(reason);
    }
  }
• Throw exception when needed:
  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:

```
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(int dx, int 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<T> extends Vector<T> // DON'T
  {
      T pop() { ... }
      void push(T item) { ... }
      ...
  }
  ```

• Bad idea: Inherit all Vector methods
• Can insert/remove in the middle of the stack
• Remedy: Use aggregation
  ```java
  public class Stack<T>
  {
      ...
      private ArrayList<T> elements;
  }
  ```
End