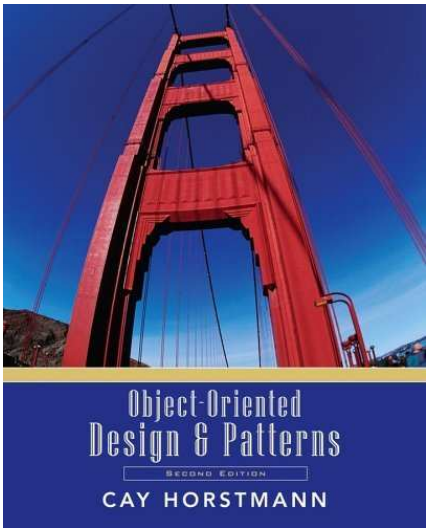
Object-Oriented Design & Patterns

Cay S. Horstmann

Chapter 6

Inheritance and Abstract Classes



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

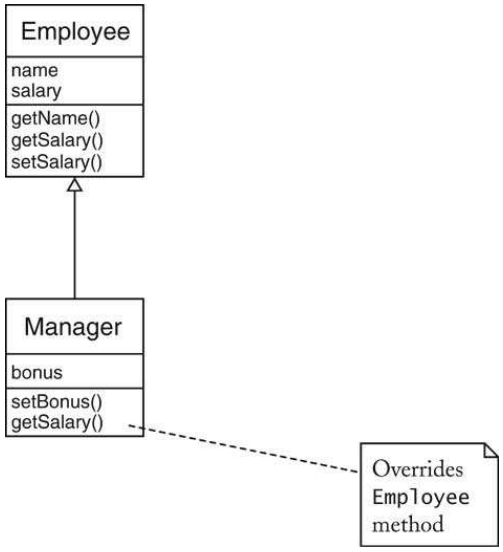
```
public class Employee
{
   public Employee(String aName) { name =
aName; }
   public void setSalary(double aSalary) {
salary = aSalary; }
   public String getName() { return name; }
   public double getSalary() { return salary;
}
   private String name;
   private double salary;
}
```

- `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



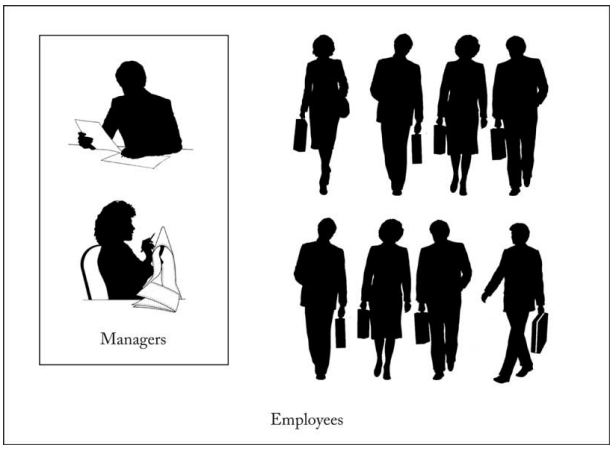
Manager Methods and Fields

- 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 **sub**class?
- 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
  ```

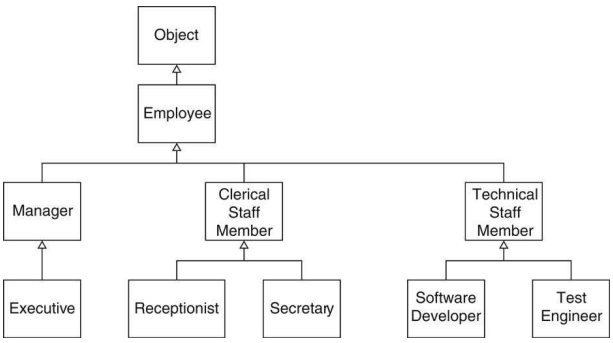
The Super/Sub Terminology



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



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

```
public class Manager extends Employee
{
   public double getSalary()
   {
      return salary + bonus;  //
ERROR--private field
   }
   ...
}
```

- Be careful when calling superclass method

```
   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:

```
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 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`

Mouse Listeners

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

Mouse Adapters

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

- 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
- Ch6/car/CarComponent.java
- Ch6/car/CarMover.java
- Ch6/car/CarShape.java


```
01: import java.awt.*;
02: import java.awt.event.*;
03: import java.awt.geom.*;
04: import javax.swing.*;
05: import java.util.*;
06:
07: /**
08:    A component that shows a scene composed of items.
09: */
10: public class CarComponent extends JComponent
11: {
12:    public CarComponent()
13:    {
14:       car = new CarShape(20, 20, 50);
15:       addMouseListener(new
16:          MouseAdapter()
17:          {
18:             public void mousePressed(MouseEvent event)
19:             {
20:                mousePoint = event.getPoint();
21:                if (!car.contains(mousePoint))
22:                   mousePoint = null;
23:             }
24:          });
25:
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:
35:                double dx = mousePoint.getX() - lastMousePoint.getX();
36:                double dy = mousePoint.getY() - lastMousePoint.getY();
37:                car.translate((int) dx, (int) dy);
38:                repaint();
39:             }
40:          });
41:    }
42:
43:    public void paintComponent(Graphics g)
44:    {
```

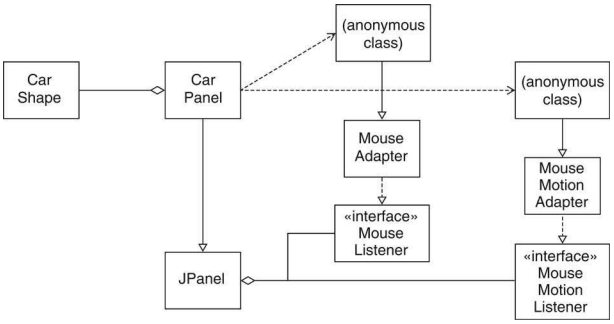
```
45:       Graphics2D g2 = (Graphics2D) g;
46:       car.draw(g2);
47:    }
48:
49:    private CarShape car;
50:    private Point mousePoint;
51: }
```

```
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:
16:       frame.add(new CarComponent());
17:       frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
18:       frame.setVisible(true);
19:    }
20:
21:    private static final int FRAME_WIDTH = 400;
22:    private static final int FRAME_HEIGHT = 400;
23: }
24:
25:
```

```
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:    }
21:
22:    public void draw(Graphics2D g2)
23:    {
24:       Rectangle2D.Double body
25:          = new Rectangle2D.Double(x, y + width / 6,
26:             width - 1, width / 6);
27:       Ellipse2D.Double frontTire
28:          = new Ellipse2D.Double(x + width / 6, y + width / 3,
29:             width / 6, width / 6);
30:       Ellipse2D.Double rearTire
31:          = new Ellipse2D.Double(x + width * 2 / 3,
32:             y + width / 3,
33:             width / 6, width / 6);
34:
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);
53:
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;
77: }
```

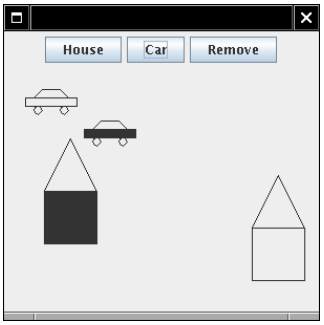

Car Mover Program



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?

The SceneShape Interface Type

SceneShape
manage selection state
draw the shape
move the shape
containment testing

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);
}
```

`CarShape` and `HouseShape` Classes

previous | start | next

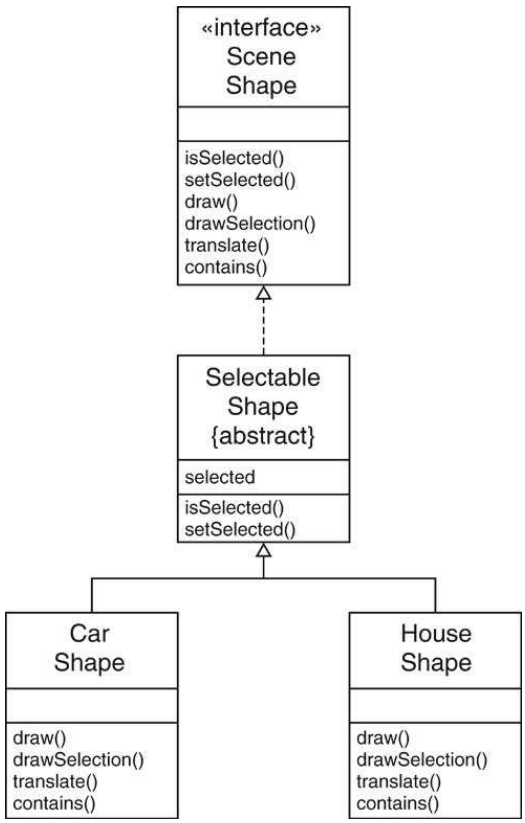
Abstract Classes

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

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

previous | start | next

Abstract Classes



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
- Ch6/scene1/SceneComponent.java
- Ch6/scene1/SceneEditor.java
- Ch6/scene1/HouseShape.java

```
01: import java.awt.*;
02: import java.awt.event.*;
03: import java.awt.geom.*;
04: import javax.swing.*;
05: import java.util.*;
06:
07: /**
08:    A component that shows a scene composed of shapes.
09: */
10: public class SceneComponent extends JComponent
11: {
12:    public SceneComponent()
13:    {
14:       shapes = new ArrayList<SceneShape>();
15:
16:       addMouseListener(new
17:          MouseAdapter()
18:          {
19:             public void mousePressed(MouseEvent event)
20:             {
21:                mousePoint = event.getPoint();
22:                for (SceneShape s : shapes)
23:                {
24:                   if (s.contains(mousePoint))
25:                      s.setSelected(!s.isSelected());
26:                }
27:                repaint();
28:             }
29:          });
30:
31:       addMouseMotionListener(new
32:          MouseMotionAdapter()
33:          {
34:             public void mouseDragged(MouseEvent event)
35:             {
36:                Point lastMousePoint = mousePoint;
37:                mousePoint = event.getPoint();
38:                for (SceneShape s : shapes)
39:                {
40:                   if (s.isSelected())
41:                   {
42:                      double dx = mousePoint.getX() - lastMousePoint.getX();
43:                      double dy = mousePoint.getY() - lastMousePoint.getY();
44:                      s.translate((int) dx, (int) dy);
```

```
45:                   }
46:                }
47:                repaint();
48:             }
49:          });
50:    }
51:
52:    /**
53:       Adds a shape to the scene.
54:       @param s the shape to add
55:    */
56:    public void add(SceneShape s)
57:    {
58:       shapes.add(s);
59:       repaint();
60:    }
61:
62:    /**
63:       Removes all selected shapes from the scene.
64:    */
65:    public void removeSelected()
66:    {
67:       for (int i = shapes.size() - 1; i >= 0; i--)
68:       {
69:          SceneShape s = shapes.get(i);
70:          if (s.isSelected()) shapes.remove(i);
71:       }
72:       repaint();
73:    }
74:
75:    public void paintComponent(Graphics g)
76:    {
77:       super.paintComponent(g);
78:       Graphics2D g2 = (Graphics2D) g;
79:       for (SceneShape s : shapes)
80:       {
81:          s.draw(g2);
82:          if (s.isSelected())
83:             s.drawSelection(g2);
84:       }
85:    }
86:
87:    private ArrayList<SceneShape> shapes;
```

```
88:    private Point mousePoint;
89: }
```

```
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 edit a scene composed
08:    of items.
09: */
10: public class SceneEditor
11: {
12:    public static void main(String[] args)
13:    {
14:       JFrame frame = new JFrame();
15:       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
16:
17:       final SceneComponent scene = new SceneComponent();
18:
19:       JButton houseButton = new JButton("House");
20:       houseButton.addActionListener(new
21:          ActionListener()
22:          {
23:             public void actionPerformed(ActionEvent event)
24:             {
25:                scene.add(new HouseShape(20, 20, 50));
26:             }
27:          });
28:
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:
39:       JButton removeButton = new JButton("Remove");
40:       removeButton.addActionListener(new
41:          ActionListener()
42:          {
43:             public void actionPerformed(ActionEvent event)
44:             {
```

```
45:                scene.removeSelected();
46:             }
47:          });
48:
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: }
60:
61:
```

```
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:
22:    public void draw(Graphics2D g2)
23:    {
24:       Rectangle2D.Double base
25:          = new Rectangle2D.Double(x, y + width, width, width);
26:
27:       // The left bottom of the roof
28:       Point2D.Double r1
29:          = new Point2D.Double(x, y + width);
30:       // The top of the roof
31:       Point2D.Double r2
32:          = new Point2D.Double(x + width / 2, y);
33:       // The right bottom of the roof
34:       Point2D.Double r3
35:          = new Point2D.Double(x + width, y + width);
36:
37:       Line2D.Double roofLeft
38:          = new Line2D.Double(r1, r2);
39:       Line2D.Double roofRight
40:          = new Line2D.Double(r2, r3);
41:
42:       g2.draw(base);
43:       g2.draw(roofLeft);
44:       g2.draw(roofRight);
```

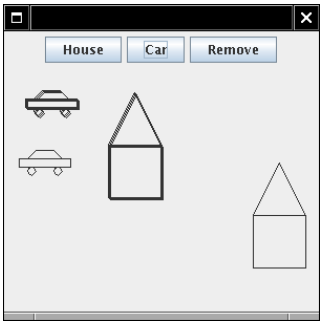
```
45:    }
46:
47:    public void drawSelection(Graphics2D g2)
48:    {
49:       Rectangle2D.Double base
50:          = new Rectangle2D.Double(x, y + width, width, width);
51:       g2.fill(base);
52:    }
53:
54:    public boolean contains(Point2D p)
55:    {
56:       return x <= p.getX() && p.getX() <= x + width
57:          && y <= p.getY() && p.getY() <= y + 2 * width;
58:    }
59:
60:    public void translate(int dx, int dy)
61:    {
62:       x += dx;
63:       y += dy;
64:    }
65:
66:    private int x;
67:    private int y;
68:    private int width;
69: }
```


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`

```
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
  `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


```
01: import java.awt.*;
02: import java.awt.geom.*;
03:
04: /**
05:    A shape that manages its selection state.
06: */
07: public abstract class SelectableShape implements SceneShape
08: {
09:    public void setSelected(boolean b)
10:    {
11:       selected = b;
12:    }
13:
14:    public boolean isSelected()
15:    {
16:       return selected;
17:    }
18:
19:    public void drawSelection(Graphics2D g2)
20:    {
21:       translate(1, 1);
22:       draw(g2);
23:       translate(1, 1);
24:       draw(g2);
25:       translate(-2, -2);
26:    }
27:
28:    private boolean selected;
29: }
```

```
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:
22:    public void draw(Graphics2D g2)
23:    {
24:       Rectangle2D.Double base
25:          = new Rectangle2D.Double(x, y + width, width, width);
26:
27:       // The left bottom of the roof
28:       Point2D.Double r1
29:          = new Point2D.Double(x, y + width);
30:       // The top of the roof
31:       Point2D.Double r2
32:          = new Point2D.Double(x + width / 2, y);
33:       // The right bottom of the roof
34:       Point2D.Double r3
35:          = new Point2D.Double(x + width, y + width);
36:
37:       Line2D.Double roofLeft
38:          = new Line2D.Double(r1, r2);
39:       Line2D.Double roofRight
40:          = new Line2D.Double(r2, r3);
41:
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(int dx, int dy)
54:    {
55:       x += dx;
56:       y += dy;
57:    }
58:
59:    private int x;
60:    private int y;
61:    private int width;
62: }
```


TEMPLATE METHOD Pattern

Context

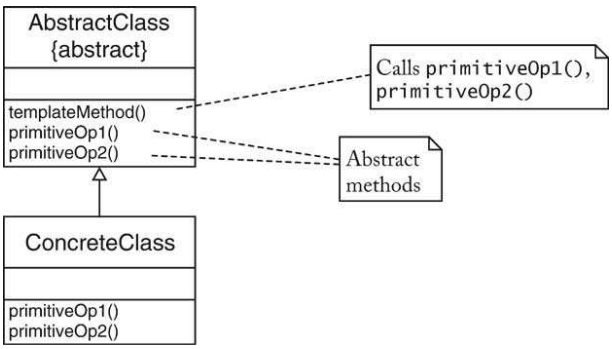
1. An algorithm is applicable for multiple types.
2. The algorithm can be broken down into *primitive operations*. The primitive operations can be different for each type
3. The order of the primitive operations doesn't depend on the type

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



TEMPLATE METHOD Pattern

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

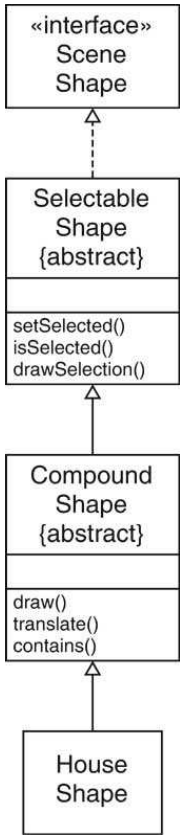
```
GeneralPath path = new GeneralPath(); path.append(new Rectangle(...), false); path.append(new Triangle(...), false); g2.draw(path);
```

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


```
01: import java.awt.*;
02: import java.awt.geom.*;
03:
04: /**
05:    A scene shape that is composed of multiple geometric shapes.
06: */
07: public abstract class CompoundShape extends SelectableShape
08: {
09:    public CompoundShape()
10:    {
11:       path = new GeneralPath();
12:    }
13:
14:    protected void add(Shape s)
15:    {
16:       path.append(s, false);
17:    }
18:
19:    public boolean contains(Point2D aPoint)
20:    {
21:       return path.contains(aPoint);
22:    }
23:
24:    public void translate(int dx, int dy)
25:    {
26:       path.transform(
27:             AffineTransform.getTranslateInstance(dx, dy));
28:    }
29:
30:    public void draw(Graphics2D g2)
31:    {
32:       g2.draw(path);
33:    }
34:
35:    private GeneralPath path;
36: }
37:
38:
39:
40:
41:
42:
43:
```

```
01: import java.awt.*;
02: import java.awt.geom.*;
03:
04: /**
05:    A house shape.
06: */
07: public class HouseShape extends CompoundShape
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:       Rectangle2D.Double base
18:          = new Rectangle2D.Double(x, y + width, width, width);
19:
20:       // The left bottom of the roof
21:       Point2D.Double r1
22:          = new Point2D.Double(x, y + width);
23:       // The top of the roof
24:       Point2D.Double r2
25:          = new Point2D.Double(x + width / 2, y);
26:       // The right bottom of the roof
27:       Point2D.Double r3
28:          = new Point2D.Double(x + width, y + width);
29:
30:       Line2D.Double roofLeft
31:          = new Line2D.Double(r1, r2);
32:       Line2D.Double roofRight
33:          = new Line2D.Double(r2, r3);
34:
35:       add(base);
36:       add(roofLeft);
37:       add(roofRight);
38:    }
39: }
```

Compound Shapes



Access to Superclass Features

- Why does the `HouseShape` constructor call `add`?

```
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
  `protected GeneralPath path; // DON'T`
- Ok to make methods protected
  `protected void add(Shape s) // GOOD`
- Protected interface separate from public interface

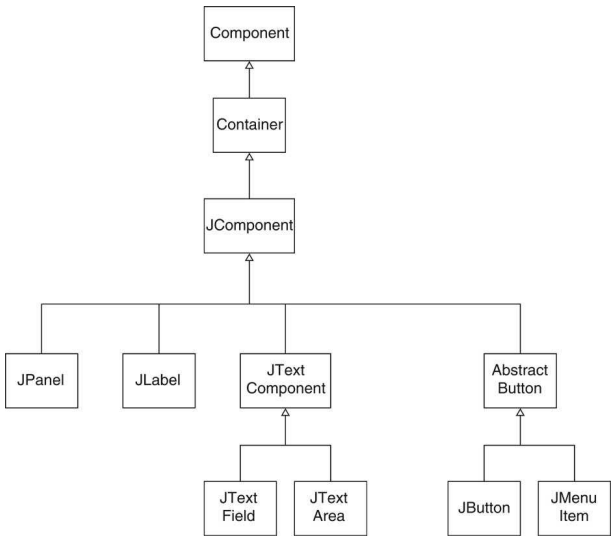
Hierarchy of Swing Components

- Base of hierarchy: `Component`
- Huge number of common methods:

```
int getWidth()int getHeight()Dimension getPreferredSize()void setBackground(Color c). . .
```

- Most important subclass: `Container`

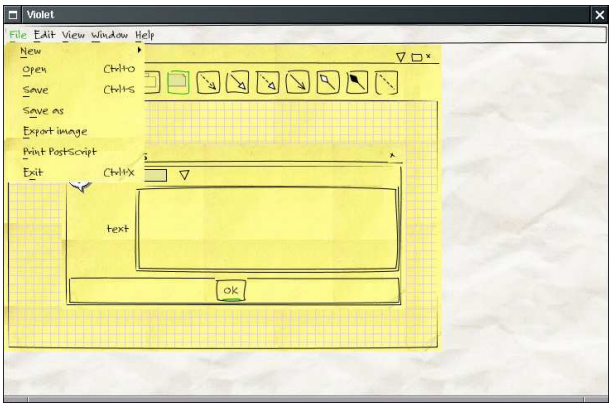
Hierarchy of Swing Components



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`, `JButton`, ... are subclasses of `JComponent`
- Intermediate classes `AbstractButton`, `JTextComponent`

Hierarchy of Geometrical Shapes

- First version of Java: few shapes, integer coordinates

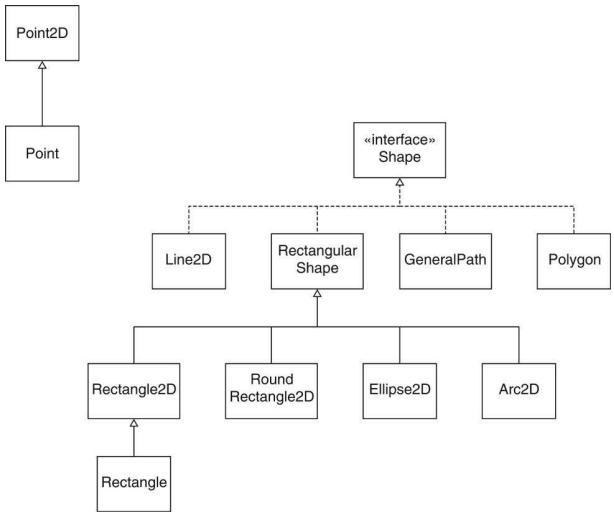
  `PointRectanglePolygon`

- Java 2: sophisticated shapes, floating-point coordinates

  `Point2DRectangle2DRoundRectangle2DLine2DEllipse2DArc2DQuadCurve2DCubicCurve2DGeneralPathArea`

- 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`

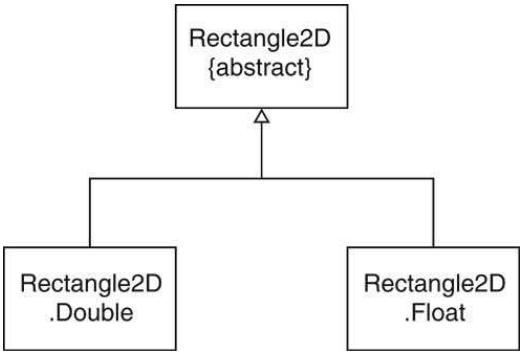
Float/Double Classes

- 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



Float/Double Classes

Float/Double Classes

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);
```

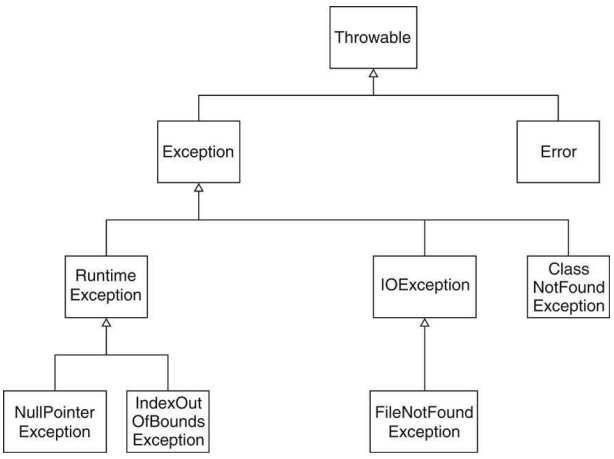
TEMPLATE METHOD Pattern

Name in Design Pattern	Actual Name (Rectangles)
AbstractClass	Rectangle
ConcreteClass	Rectangle2D.Double
templateMethod()	contains
primitiveOp*n*()	getX, getY, getWidth, getHeight

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:

```
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:

```
public class Point {    public Point(int xm, int ym) { ... }    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:

```
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:

```
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

```
public class Stack<T>{   ...   private ArrayList<T> elements;}
```