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

6.1 Concept of Inheritance

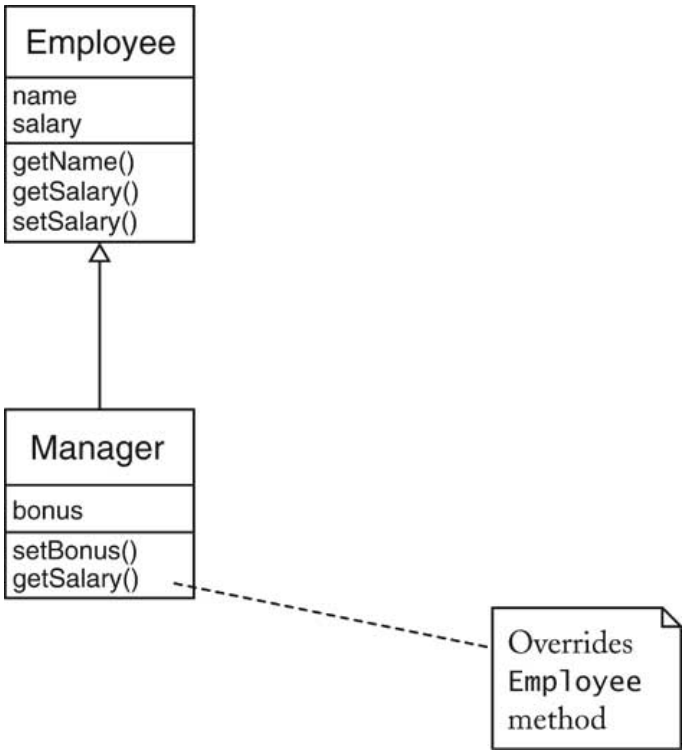
6.1.1 Using Inheritance for Modeling Specialization

- `Employee` class

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

- `Manager` is a subclass

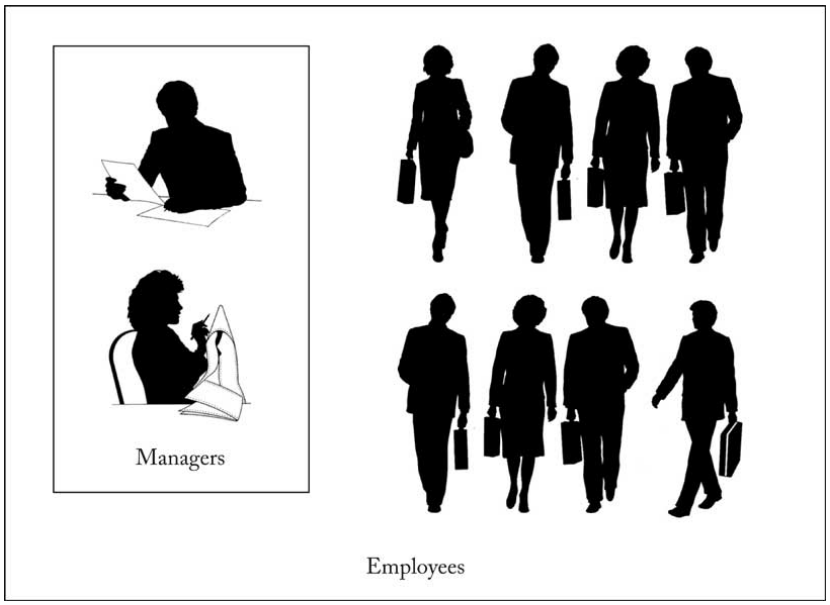
```
public class Manager extends Employee
{
   private double bonus; // new field


   public Manager(String aName) { ... }

   // overridden method
   public double getSalary() { ... }

   // new method
public void setBonus(double aBonus) { bonus = aBonus; }     }
```



- Manager Methods and Fields
  - fields name and salary (inherited from Employee)
  - field bonus (newly defined in Manager)
  - methods setSalary, getname (inherited from Employee)
  - method getSalary (overridden in Manager)
  - method setBonus (newly defined in Manager)

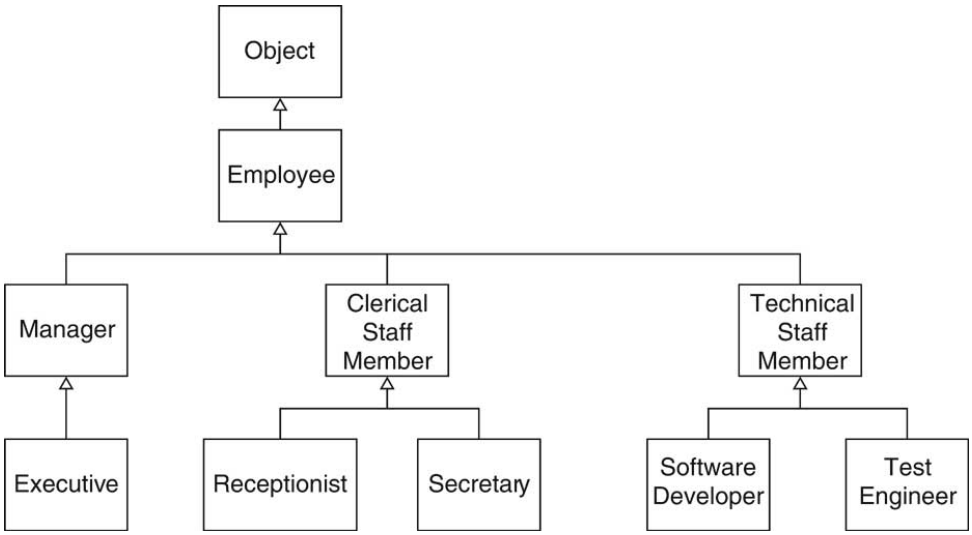
6.1.2 The Super/Sub Terminology

Why is Manager a **sub**class? The set of managers is a *subset* of the set of employees



6.1.3 Inheritance Hierarchies

- Programming: Inheritance hierarchy
  - General superclass at root of tree
  - More specific subclasses are children



6.1.4 The Substitution Principle (Liskove substitution principle)

- The principle is based on "is-a relationship" between classes.

- You can use a subclass object B whenever a super class object A is expected because B *is* A.
- Example: Can set `e` to `Manager` reference because a manager is an employee.

```
public void someMethod(Employee e)
{   System.out.println("salary=" + e.getSalary());}
```

- Polymorphism: Correct `getSalary` method is invoked based on the actual object the variable e references not the type of the variable e.

6.1.5 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
}
```

- Use `super` keyword

```
public double getSalary()
{
   return super.getSalary() + bonus;
}
```

- `super` suppresses the polymorphic call mechanism and forces the superclass method to be called instead. It calls an appropriate method of the closest class in the hierarchy.

6.1.6 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, a super() call is implicit.
- Calls an appropriate constructor of the immediate super class. If the immediate super class does not have the expected constructor, a compilation error is generated.

Example:

```
public class Computer
{    private String manufacturer;
     private String processor;
     private int ramSize;
     private int diskSize;

    public Computer(String man, String processor, int ram, int disk)
    {    manufacturer = man;
         this.processor = processor;    // not recommended. Use a
different name !
         ramSize = ram;
         diskSize = disk;
    }

   // Other Methods are defined here.

   public String toString()
   {    String result = "Manufacturer: " + manufacturer +
            "\nCPU: " + processor +
            "\nRAM: " + ramSize + " megabytes" +
            "\nDisk: " + diskSize + " gigabytes";
        return result;
  }
}
public class LapTop extends Computer
{
  // Data Fields
  private static final String DEFAULT_LT_MAN = "MyBrand";
  private double screenSize;
  private double weight;
  public LapTop(String man, String proc, int ram, int disk,
                double screen, double wei)
  {
    super(man, proc, ram, disk);
    screenSize = screen;
    weight = wei;
  }
```

```
  /** Initializes a LapTop object with 5 properties specified. */
  public LapTop(String proc, int ram, int disk,double screen, double
wei)
  {
    this(DEFAULT_LT_MAN, proc, ram, disk, screen, wei);
  }

  public String toString()
  {
    String result = super.toString() + "\nScreen size: " + screenSize
+ " inches" + "\nWeight: " + weight + " pounds";
    return result;
  }
}
```

1)

6.1.7 Preconditions and Postconditions of Inheritied Methods

- A subclass method can only require a *precondition that is at most as strong as* the precondition of the inherited method that it overrides.

```
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

```
public void someMethod(Employee e)
{  e.setSalary(50000); }
```

The above program appears to be correct because the method parameter is > 0, fulfilling the precondition of the Employee method. However, if e refers to a Manager object at run time, the precondition that the manager's salary should be at least 100000 is violated.

- *A sublcass method must ensure a post condition that is at least as strong as* the post condition of the method that it overrides. For example, if the setSalary method of the Employee class promises not to decrease salary, all methods that override it, such as the setSalary method of the Manager class, must make the same promises or a stronger promise.

6.1.8 Notes

Adding responsibilities: Decorator vs. Inheritance

Sometimes we want to add responsibilities to individual objects, not to an entire class. A graphical user interface toolkit, for example, should let you add properties like borders or behaviors like scrolling to any user interface component.

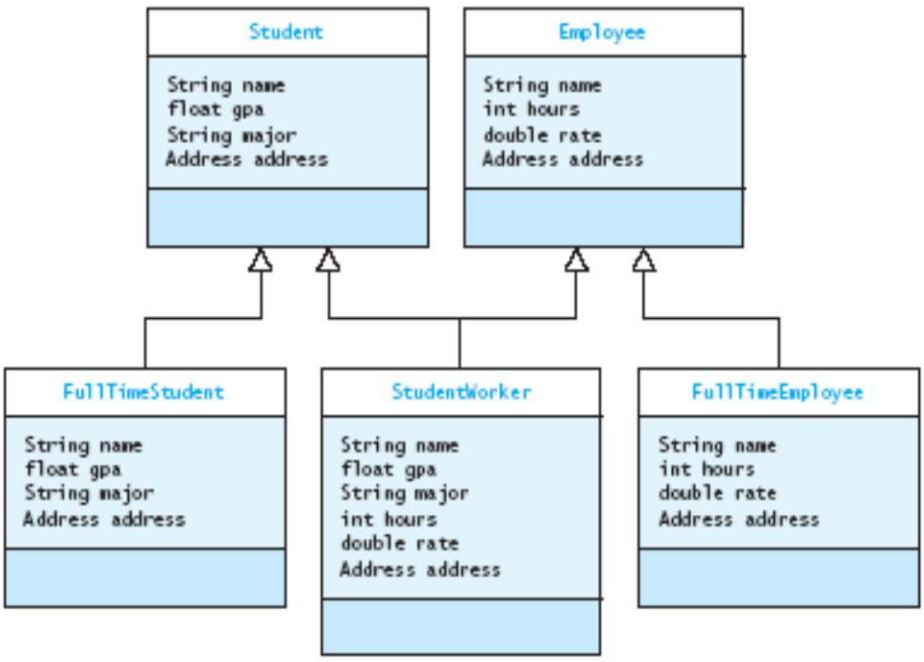
- One way to add responsibilities is with inheritance. Inheriting a border from another class puts a border around every subclass instance. This is inflexible, however, because the choice of border is made *statically*. A client can't control how and when to decorate the component with a border.
- Another way is to enclose the component in another object that adds the border. The enclosing object is called a **decorator**. The decorator conforms to the interface of the component it decorates so that its presence is transparent to the component's clients. Transparency lets you nest decorators recursively, thereby allowing an unlimited number of added responsibilities.

Multiple Inheritance

In Java, a class can extend *at most* one immediate super class. Why ?

- Multiple inheritance: the ability to extend more than one class
- Multiple inheritance is a language feature that is difficult to implement and can lead to ambiguity. Therefore, Java does *not allow* a class to extend more than one class. Use multiple interfaces to emulate multiple inheritance.

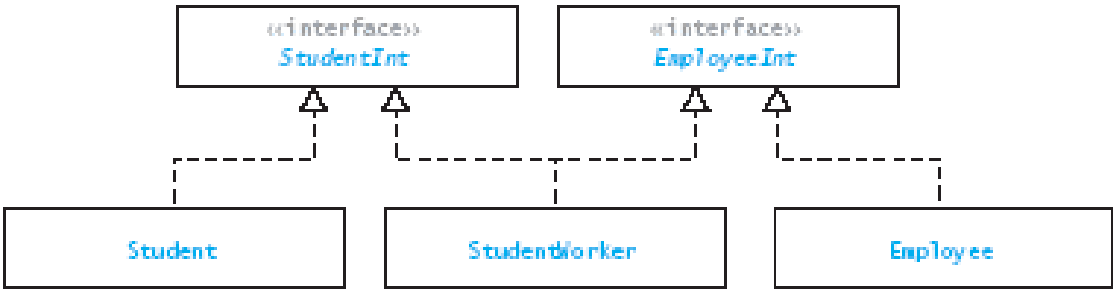
FIGURE 3.9
Class StudentWorker Extends Student and Employee



Use multiple interfaces to emulate multiple inheritance, however, …

- A problem of *version control*: if we make any change to a method in class Student, we have to remember to make the same change to the method in class StudentWorker.

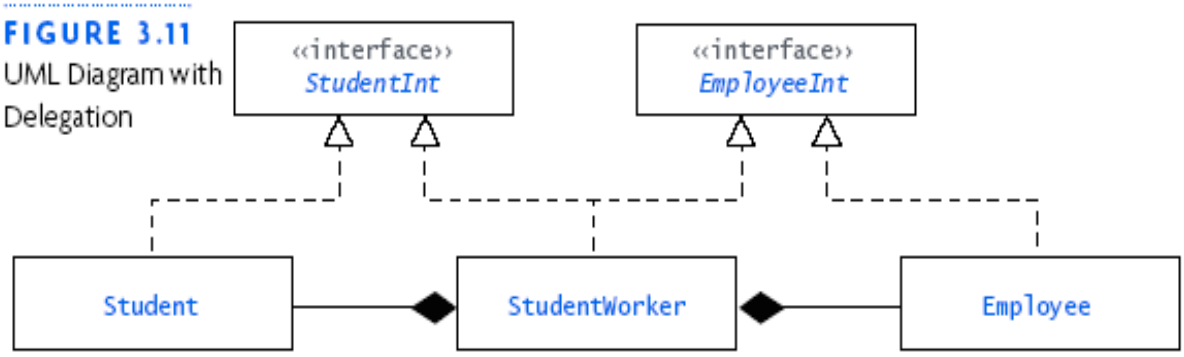
FIGURE 3.10
Class Hierarchy with Interfaces StudentInt and EmployeeInt



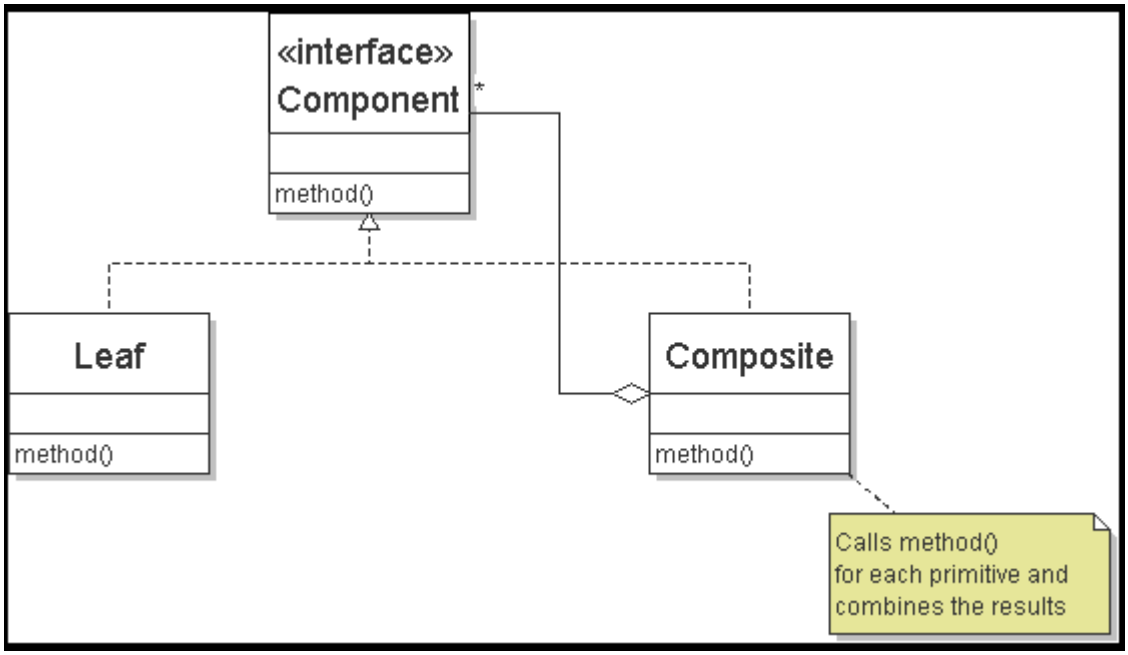
Delegation

- You can reduce duplication of modifications and reduce problems associated with version control through a technique known as delegation
- In delegation, a method of one class accomplishes an operation by delegating it to a method of another class

```
public class StudentWorker implements StudentInt, EmployeeInt
{    private Student student;
     private Employee employee;

     public StudentWorker(String nam, double grade, String maj,
                          double hour, double rat, Address addr)
     {    student = new Student(nam, grade, maj, addr);
          employee = new Employee(nam, hour, rat, addr);
     }

     public String getName() { return student.getName();  }
     public double getGPA() { return student.getGPA(); }
     public String getMajor() { return student.getMajor(); }
     public Address getAddress() { return student.getAddress(); }
     public double getHours() {  return employee.getHours(); }
     public double getRate() {  return employee.getRate(); }
}
```



Example of delegation we already saw: Composite Pattern (Chapter 5)



6.2 Graphic Programming with Inheritance

6.2.1 Designing Subclasses of the JComponent Class

- Create drawings by forming a subclass of JComponent

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

- Advantage: Inherit behavior from JComponent. For example, the CarComponent class inherits the `addMouseListener` and `addMouseMotionListener` from the JComponent class to attach mouse listeners. (In fact, these methods are originally defined in Component, a super class of JComponent.)

6.2.2 Listener Interface Types and Adapter Classes

Listener Interfaces

- There are two different listener types: MouseListener and MouseMotionListener.

```
public interface MouseListener extends EventListener
{  void mouseClicked(MouseEvent event);
   void mousePressed(MouseEvent event);
   void mouseReleased(MouseEvent event);
   void mouseEntered(MouseEvent event);
   void mouseExited(MouseEvent event);
}
public interface MouseMotionListener extends EventListener
{  void mouseMoved(MouseEvent event);
   void mouseDragged(MouseEvent event);
}
```

- To add a listener, call the addMouseListener or addMouseMotionListener method with a listener object, an instance of a class that implements the listener interface.

```
public void addMouseListener(MouseListener l)
public void addMouseMotionListener(MouseMotionListener l)
```

Adapter classes

After Java 7.0, MouseAdapter implements both MouseListener and MouseActionListener.

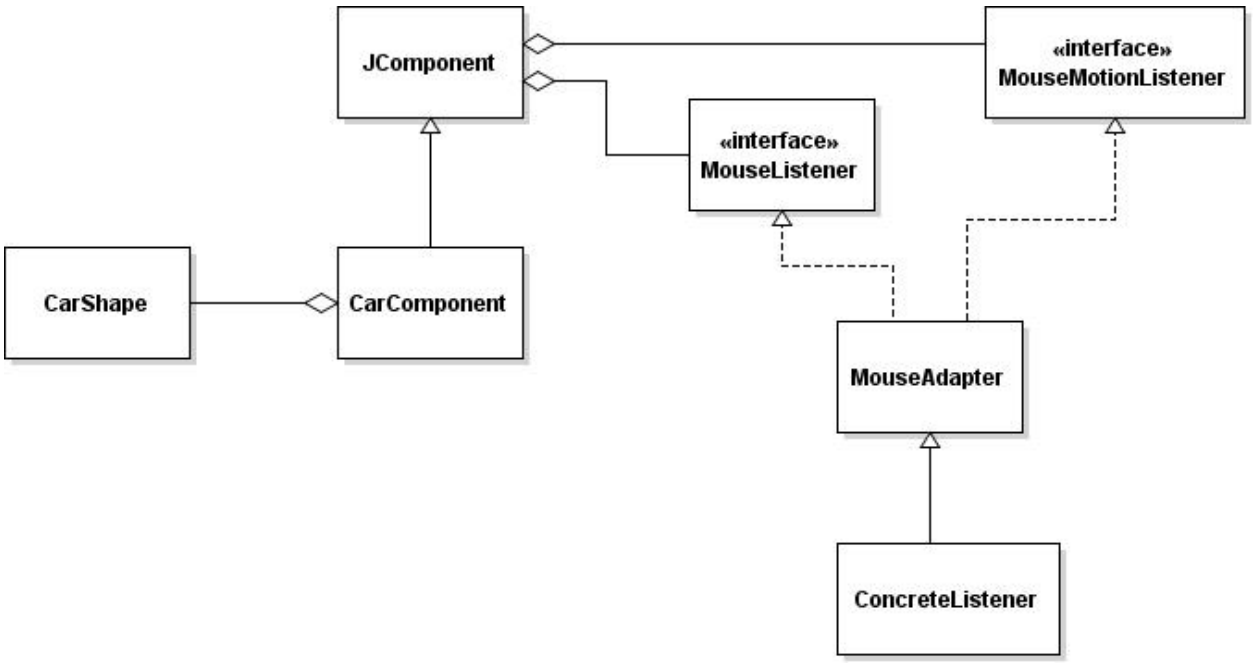
- A class that implements an interface must provide all the method defined in the interface. An adapter class simplifies the implementations of listeners.

```
public abstract class MouseAdapter
           implements MouseListener, MouseWheelListener, MouseMotionListener
{
public void mouseClicked(MouseEvent e){};
public void mouseDragged(MouseEvent e){};
public void mouseEntered(MouseEvent e){};
public void mouseExited(MouseEvent e){};
public void mouseMoved(MouseEvent e){};
public void mousePressed(MouseEvent e){};
public void mouseReleased(MouseEvent e){};
public void mouseWheelMoved(MouseEvent e){};
}
```

- The API reads "create a listener object using the extended class and then register it with a component using the component's addMouseListener, addMouseMotionListener, addMouseWheelListener methods".

```
private class MouseListeners extends MouseAdapter
{
    public void mousePressed(MouseEvent e) {. . . }

    public void mouseDragged(MouseEvent e) {. . . }
}


MouseListeners listeners = new MouseListeners();
addMouseListener(listeners);
addMouseMotionListener(listeners);
```

Example: Car Mover Program



- Ch6/car/CarComponent.java
  - A MouseListener and a MouseMotionListener are attached to the Car Component class. You need one concrete class that extends the MouseAdapter class.
  - When a mouse button is pressed/dragged, a MouseEvent are generated and sent to this registered MouseListeners/MouseMotionListener.
  - The mousePressed method of the MouseListener remembers the position at which the mouse was pressed: event.getPoint().
  - The mouseDragged method translates car shape.

```
/**
   A component that shows a scene composed of items.
*/
public class CarComponent extends JComponent
{
   private CarShape car;
   private Point mousePoint;

   public CarComponent()
   {
      car = new CarShape(20, 20, 50);
      MouseListeners listeners = new MouseListeners();
      addMouseListener(listeners);
      addMouseMotionListener(listeners);
   }

   private class MouseListeners extends MouseAdapter
```

```
   {

            public void mousePressed(MouseEvent event)
            {
               mousePoint = event.getPoint();
               if (!car.contains(mousePoint))
                  mousePoint = null;
            }
         public void mouseDragged(MouseEvent event)
            {
               if (mousePoint == null) return;
               Point lastMousePoint = mousePoint;
               mousePoint = event.getPoint();

           double dx = mousePoint.getX() - lastMousePoint.getX();
           double dy = mousePoint.getY() - lastMousePoint.getY();
               car.translate((int) dx, (int) dy);
               repaint();
            }

   }

   public void paintComponent(Graphics g)
   {
      Graphics2D g2 = (Graphics2D) g;
      car.draw(g2);
   }
}
```

- Ch6/car/CarMover.java – Test program
- Ch6/car/CarShape – The class which represents the car shape itself.

6.3 Abstract Classes

6.3.1 Abstract basics

- A class must be declared as abstract if it has an abstract method.
- A class C has abstract methods if any of the following is true:
  - C explicitly contains a declaration of an abstract method.
  - Any of C's super classes has an abstract method and C neither declares nor inherits a method that implements it.
  - A direct super interface of C declares or inherits a method, and C neither declares nor inherits a method that implements it.

- An abstract class can have instance variables, constructors, and concrete methods. The constructors of an abstract class will be used by its subclasses to initialize inherited instance variables.
- An abstract class cannot be instantiated, however, can define a type.

```
Point p = new Point(2,3); // compilation error
Point p = new SimplePoint(2,3); // correct
```

```
abstract class Point
{    int x;  int y;
     Point (int pX, int pY)
     { x = pX; y = pY; }
     void move(int dx, int dy)
     {   x += dx;
         y += dy;
     }
     abstract void alert();
}
abstract class ColoredPoint extends Point
{
   int color;
   ColoredPoint(int pX, int pY, int pC)
   { super (pX, pY);
     color = pC;
   }
}
class SimplePoint extends Point
{  SimplePoint(int pX, int pY)
   {  super (pX, pY); }
   void alert() { }
}
```

6.3.2 Final classes

- final class
  - cannot be extended.
  - final is implicit for all methods of a final class.
  - a final class cannot have abstract methods.
- final method
  - cannot be overriden in a subclass.
  - cannot be abstract.
- Why final classes & methods ?
  - To ensure that the method or class is used only as you wrote it, and not through an overriding version in a subclass.

```
final class FinalClass
```

```
{  protected void protetedMethod()
   {  }
}

/* This doesn't compile.
class FinalSubclass extends FinalClass
{ }
*/

class ClassWithFinalMethods
{  public static final void staticFinalMethod()
   { }
}

class SubclassWithFinalMethods extends ClassWithFinalMethods
{   /*  This doesn't compile.
       public static void staticFinalMethod()
       {   }
   */
}
```

6.3.3 Intent of using abstract classes

An abstract class is being used in a class hierarchy when we need a base class for two or more actual classes that share some attributes and/or some behaviors. Consider the following example to learn the use of abstract classes.

- The `SceneShape` Interface type that defines a number of operations that the shapes must carry out.

```
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);
}
```

- Abstract classes are convenient placeholders for factoring out common behavior. Consider the following example to learn the use of abstract classes.

```
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; }

}
```

- It is apparent that classes that implement SceneShape will have the above three lines in common. Therefore, you can factor out these common behaviors and put them in a class. Subclasses of the class will inherit these common behaviors. Some of the methods of SceneShape, such as draw, drawSelection, translated, contains, should be specific to subclasses, and therefore, are defined as abstract methods. The class is abstract if there is an abstract method defined.

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

}
```

- A class type should be declared abstract only if the intent is that subclasses can be created to complete the implementation. If the intent is simply to prevent instantiation of a class, the proper way to express this is to declare a constructor of no argument, make it private, never invoke it, and declare no other variables. A class of this form usually contains static methods and static variables.

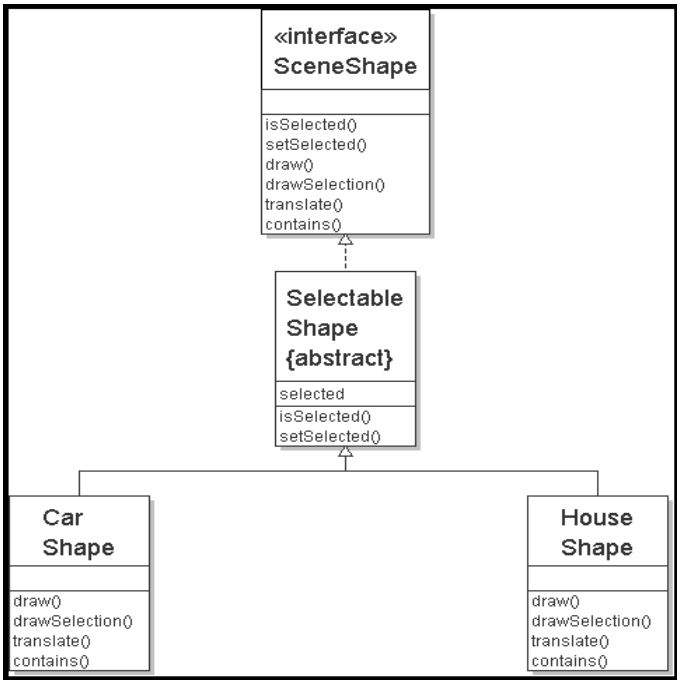
```
public final class Math
{   private Math() { };
     …
}
```

6.3.4 Comparison of concrete classes, abstract classes, and interfaces

TABLE 3.1
Comparison of Actual Classes, Abstract Classes, and Interfaces

Property	Actual Class	Abstract Class	Interface
Instances (objects) of this can be created	Yes	No	No
This can define instance variables and methods	Yes	Yes	No
This can define constants	Yes	Yes	Yes
The number of these a class can extend	0 or 1	0 or 1	0
The number of these a class can implement	0	0	Any number
This can extend another class	Yes	Yes	No
This can declare abstract methods	No	Yes	Yes
Variables of this type can be declared	Yes	Yes	Yes

Example: Scene Editor



- Ch6/scene1/SceneEditor.java

```
/**
   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));
            }
         });

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

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

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

      frame.add(scene, BorderLayout.CENTER);
      frame.add(buttons, BorderLayout.NORTH);
      frame.setSize(300, 300);
      frame.setVisible(true);
   }
}
```

- Ch6/scene1/SelectableShape.java

```
public abstract class SelectableShape implements SceneShape
{  private boolean selected;

   public void setSelected(boolean b)
   {  selected = b; }

   public boolean isSelected()
   {  return selected; }

}
```

- Ch6/scene1/HouseShape.java

```
/**
   A house shape.
*/
public class HouseShape extends SelectableShape
{  private int x;
   private int y;
   private int width;
   /**
      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);

      Point2D.Double r1 = new Point2D.Double(x, y + width);
      Point2D.Double r2 = new Point2D.Double(x + width / 2, y);
      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;
      }


   }
```

- Ch6/scene1/SceneComponent.java

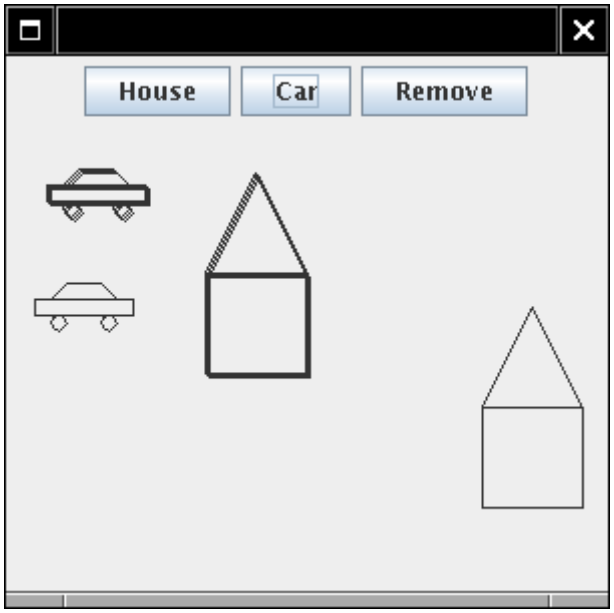
```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;
import java.util.*;

/**
   A component that shows a scene composed of shapes.
*/
public class SceneComponent extends JComponent
{
   private ArrayList<SceneShape> shapes;
   private Point mousePoint;

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

      MouseListeners listeners = new MouseListeners();
      addMouseListener(listeners);
      addMouseMotionListener(listeners);
   }

private class MouseListeners extends MouseAdapter
{
      public void mousePressed(MouseEvent event)
            {
               mousePoint = event.getPoint();
               for (SceneShape s : shapes)
               {
                  if (s.contains(mousePoint))
                     s.setSelected(!s.isSelected());
               }
               repaint();
            }
            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);
      }
   }

}
```

6.4 The TEMPLATE METHOD Pattern

- In the previous example, concrete SelectableShape classes such as Car and House should define their own drawSelection method.
- A better approach is to define a drawSelection method that can be applied to any SelectableShape class. The following method that shifts, draws, shifts, draws, restores to original position, will mark the selected shape by thickening the image of the shape.
- Ch6/scene2/SelectableShape.java

```
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);
}

public abstract class SelectableShape implements SceneShape
{
    private boolean selected;

    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);
   }
}
```



Intent

- Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- The Template Method pattern should be used
  - to implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.
  - when common behavior among subclasses should be factored and localized in a common class to avoid code duplication.

Context

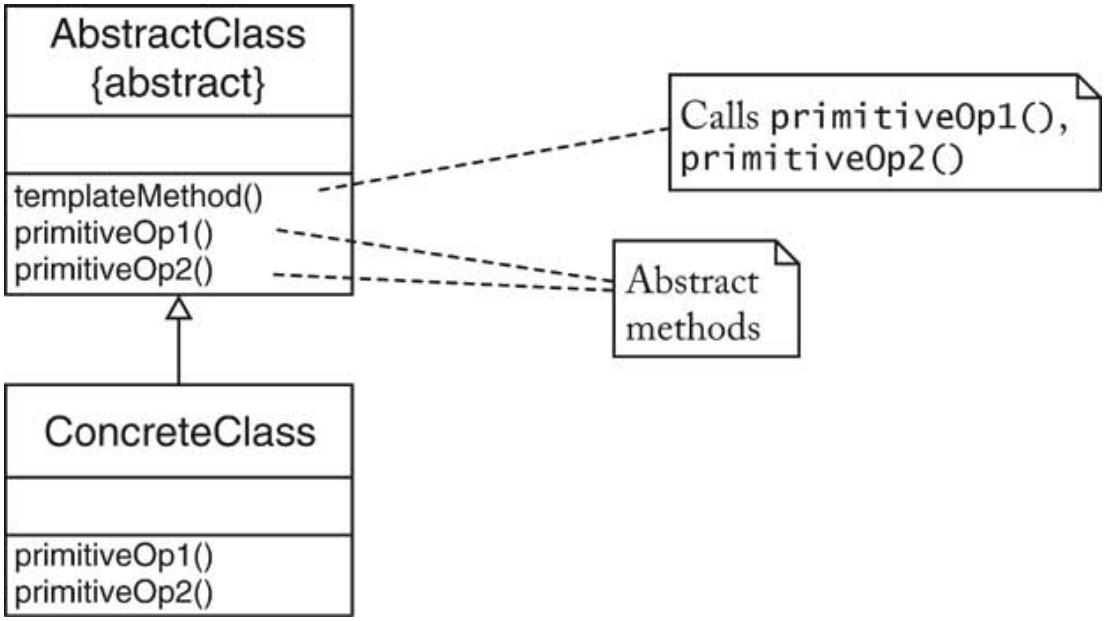
1. An algorithm is applicable for multiple types.
2. The algorithm can be broken down into *primitive operations.* The implementation of primitive operations can be different for each type.
3. The order of the primitive operations in the algorithm doesn't depend on the type.

Solution

1. Define a super class 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 complete 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.

Structure



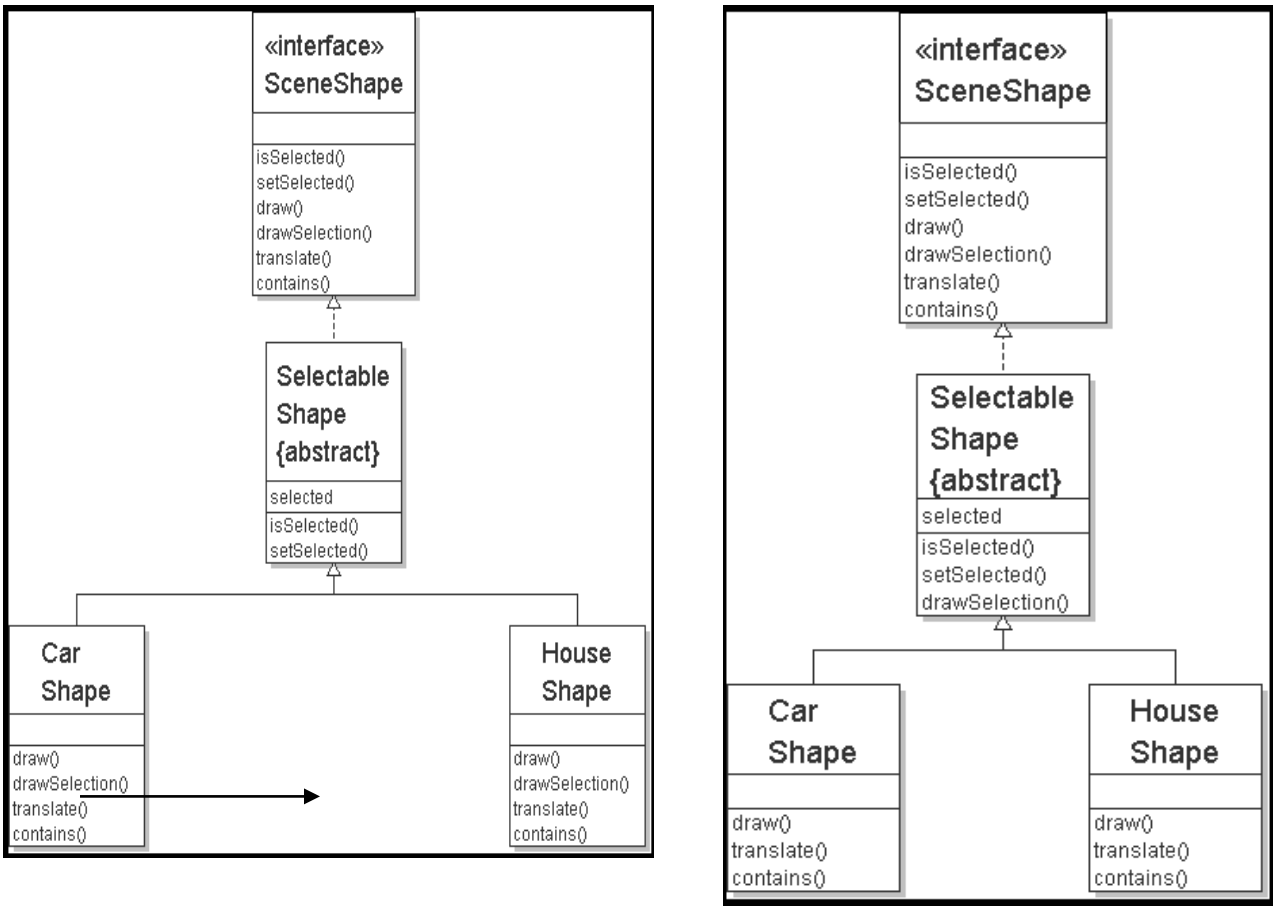
Example: the drawSelection method defined in the abstract class SelectableShape

Name in Design Pattern	Actual Name (Selectable shapes)
`AbstractClass`	`SelectableShape`
`ConcreteClass`	`CarShape, HouseShape`
`templateMethod()`	`drawSelection`
`primitiveOp1(), primitiveOp2()`	`translate, draw`

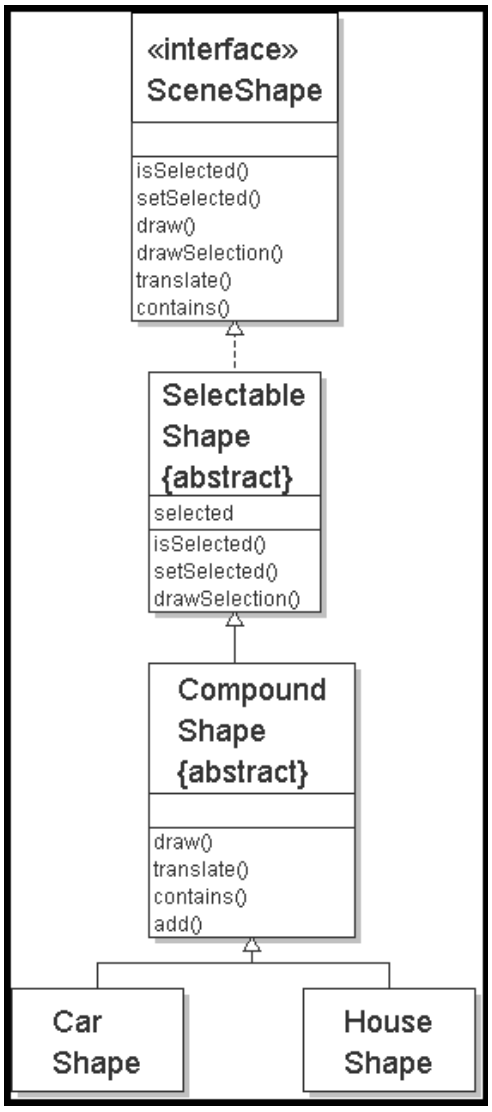
6.5 `protected` visibility

In this section, we will create one more super class called CompoundShape to move the common features of house and car shapes to this super class.

Using Template method pattern



Using CompoundShape



6.5.1 Use of GeneralPath

Note: `GeneralPath` is a legacy final class which exactly implements the behavior of its superclass Path2D.Float. Therefore, in the following lecture note,
GeneralPath path = new GeneralPath() can be replaced with Path2D path = new Path2D.Float();

Both Car and House are compound shapes. Therefore, we can create a base class that represents a compound class and move the common features to this base case. There can be many different ways to implement such a compound shape. In this section, we will use GeneralPath to store the primitive shapes of a compound shape.

1) java.awt.geom.GeneralPath

A general path is a sequence of shapes.

2) To add shapes to a GeneralPath, use the append method

```
public void append(Shape s,  boolean connect)
```
appends the geometry of the specified Shape object to the path, possibly connecting the new geometry to the existing path segments with a line segment.

Parameters:
s - the Shape whose geometry is appended to this path
connect - a boolean to control whether or not to turn an initial moveTo segment into a lineTo segment to connect the new geometry to the existing path

Example

```
GeneralPath path = new GeneralPath();
→ Path2D path = new Path2D.Double();
path.append(new Rectangle(...), false);
path.append(new Triangle(...), false);
g2.draw(path);
```

3) Advantage of using GeneralPath: Containment test is free

```
path.contains(aPoint); // tests if the path contains the given point
```

6.5.2 CompoundShape

- The *delegation* technique is adopted in the class CompoundShape
  - A ComoundShape **has** a GeneralPath that collects the primitive shapes of this compound shape.
  - The class CompoundShape **delegates** the methods of the SceneShape interface to a GeneralPath.

- Ch6/scene3/CompoundShape.java

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

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

   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); }
}
```

- The use of CompoundShape class simplifies the HouseShape class.

```
public class HouseShape extends CompoundShape
{
   public HouseShape(int x, int y, int width)
   {
      Rectangle2D.Double base = new Rectangle2D.Double(x, y + width, width, width);
      Point2D.Double r1 = new Point2D.Double(x, y + width);
      Point2D.Double r2 = new Point2D.Double(x + width / 2, y);
      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);
   }
}
```

6.5.3 Protected visibility

As part of design decision, we want to allow only compound shapes, classes that extend CompoundShape, to add their primitive shapes to the GeneralPath.

1) Could we declare the `path` instance field as `protected` ? No !

```
public abstract class CompoundShape
{   protected GeneralPath path;  // DON'T

}
```

It is a bad idea to make fields protected. After it is defined, its definition cannot be modified because some subclass might rely on it.

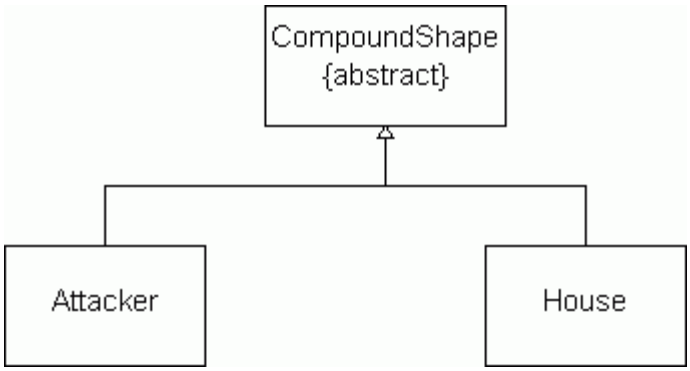
2) Can we declare the `add` method as `protected` ? Yes !

- Now, the add method can be called by subclass methods such as the HouseShape constructor. (You should be aware of that classes in the same package also have access to protected features, even if they don't belong to subclasses.)

- More restriction: you can use protected features only on objects of their own class. For example,

```
public class Attacker extends CompoundShape
{   void uglify(HouseShape house)
    { ...
house.add(aShape); // Won't work - can only call add on other Attacker
objects.
     }
}
```

A protected member of class X can be accessed by any object that *is* X. Both Attackers and Houses inherit and can use the protected add method of the CompoundShape because they are CompoundShapes. However, an Attacker cannot use the add method of House because an Attacker is not a House.



6.5.4 Summary of kinds of visibility

Visibility	Applied to Classes	Applied to Class Members
`private`	Applicable to inner classes. Accessible only to members of the class in which it is declared.	Visible only within this class.
Default or package (no Java keyword for this)	Visible to classes in this package	Visible to classes in this package.
`protected`	Applicable to inner classes. Visible to classes in this package and to classes outside the package that extend the class in which it is declared.	Visible to classes in this package and to classes outside the package that extends this class.
`public`	Visible to all classes.	Visible to all classes. The class defining the member should also be public.

6.6 Hierarchy of Swing Components

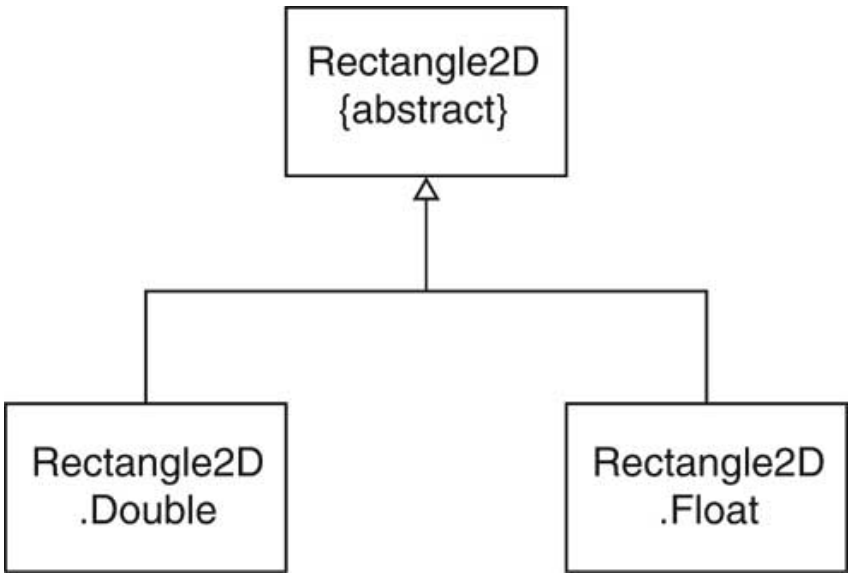
This topic was covered in the Chapter 4.

6.7 Hierarchy of Geometrical Shapes

This topic was covered in the Chapter 4. In this section, we will learn the use of template method pattern in the design of shape classes.

6.7.1 Float/Double Classes

- Each RectangleShape class has two subclasses, e.g.
  - Rectangle2D.Double/Rectangle2D.Float
  - RoundRectangle2D.Double/RoundRectangle2D.Float
  - Ellipse2D.Double/Ellipse2D.Float
- These logically related classes reside inside as static inner classes of Rectangle2D.
- Library designers provides the programmers with two different versions (.Double and .Float)
  - An object of .Float version saves memory space.
    - A float value occupies 32 bits while a double value occupies 64 bits.
    - However, note that most of intermediate computations are done in double precision.
  - Some application may need double precision.




```
public abstract class Rectangle2D extends RectangularShape
{
  . . .
```

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

    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 static class Double extends Rectangle2D
  {
    public double x;
    public double y;
    public double width;
    public double height;

    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;
    }
    ...
  }
  ...
}
```

6.7.2 Use of TemplateMethod Pattern

- The Rectangle2D defines most of useful methods shared by its subclasses: Template Method Pattern

```
public abstract class Rectangle2D extends RectangularShape
{
   ...

   public boolean contains(double x, double y)
   {
      double x0 = getX();
      double y0 = getY();
      return x >= x0 && y >= y0 && x < x0 + getWidth() && y < y0 +
getHeight();
   }
}
```

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

- Consider the following code:

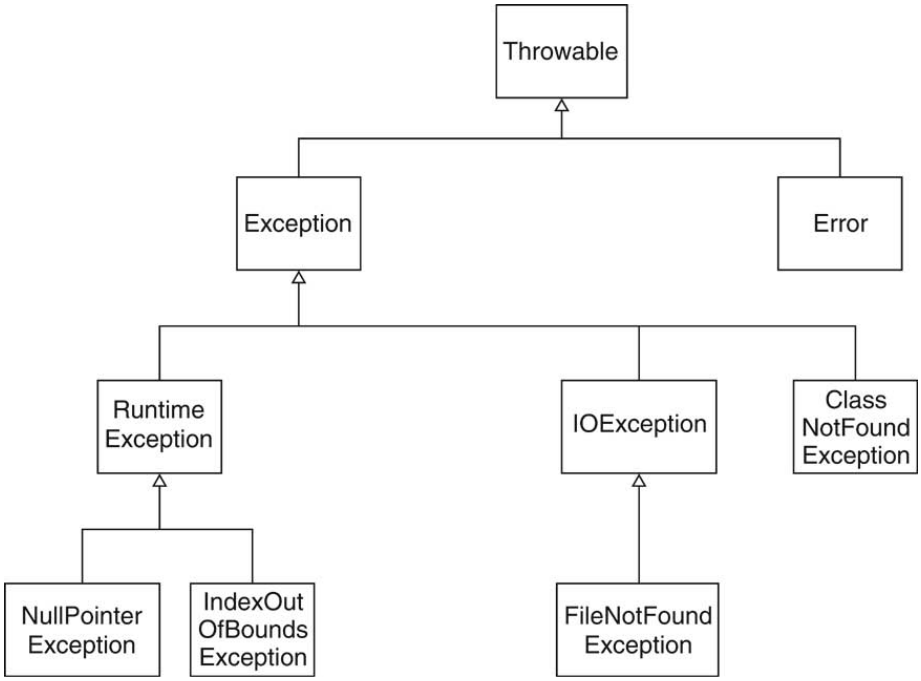
```
Rectangle2D rect = new Rectangle2D.Double(5, 10, 20, 30); // upcasting
```

Later, rect.contains(a, b) will subsequently call the getX(), getY(), getWidth(), and getHeight() method of Rectangle2D.Double class (polymorphism).

6.8 Hierarchy of Exception Classes

6.8.1 Hierarchy of Runtime-problem related Classes

- Base of hierarchy: Throwable
- Two subclasses: Error, Exception
  - Subclasses of Error: fatal errors (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)



A method has a line that throws a checked exception, the method must either handle it (try/catch) or pass it on (throws).

6.8.2 Detecting and throwing an exception

There are two entities that detect and throw an exception: Method and Virtual Machine

- Method

```
e.g.) if (amount < 0)
        throw new IllegalArgumentException();
```

- Virtual Machine

```
e.g.) Rectangle box;
        box.translate(10,15);
```

6.8.3 Exception Handling

- try and catch
- throws (to propagate the responsibility to handle an exception to the caller of the method)

(1) try and catch

- Syntax

```
class MultiCatch
{  public static void main (String [] args)
```

```
    {   try
     {  int a = args.length;
        System.out.println("a = " + a);
        int b = 42 / a;
        int [] c = { 5 };
        c[42] = 99 ;
     }
     catch (ArithmeticException e)
     { System.out.println("Divided by 0: " + e);}
     catch (ArrayIndexOutOfBoundsException e)
     { System.out.println("Array index oob: " + e);}
     finally
     { System.out.println("This code is executed in anyway");}

System.out.println(“xx”);
  }
}
```

- An exception handler recovers the problem, and has the program continues to run and exit in a normal mode.

```
import java.util.Random;
public class HandleError
{   public static void main (String [] args)
    {int a = 0, b = 0, c = 0;
     Random r = new Random();
     for (int i = 0 ; i < 100; i++)
     {  try
        {  b = r.nextInt();
           c = r.nextInt();
           a = 12345/ (b/c);
        }
        catch (ArithmeticException e)
        { System.out.println("Division by zero.");
         a = 0 ; // set a to zero and move on
        }
        System.out.println("a: " + a);
     }
   }//main
}
```

- When you use multiple catch statements, ensure that catch blocks handling subclasses come before those handling their superclasses. Note that a catch statement can handle the specified class as well as it subclasses.

```
class SuperSubCatch
```

```
{ public static void main(String [] args)
  {  try
     {   int a = 0;
         int b = 42/a;
     }
     catch (Exception e)
     {   System.out.println("Generic Exception catch."); }
     catch (ArithmeticException e)
     {   System.out.println("This is never reached."); }
  }
}
```

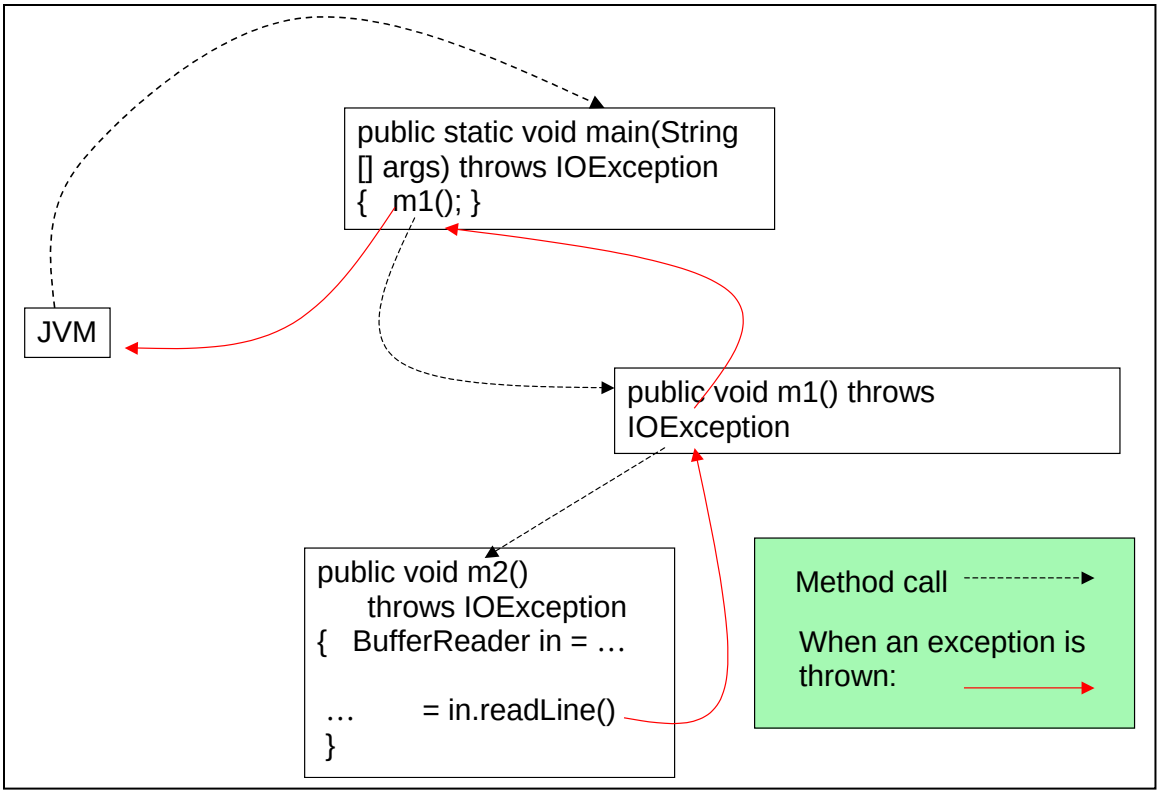
(2) throws

- If a method has a line that throws a checked exception, the method MUST either handle it (try/catch) or pass it on (throws).
- Which of the two is better ?
  - You should catch those exceptions that you know how to handle.
  - Propagate those that you don't know how to handle. In this case, you must add a throws modifier to alert the caller that an exception may be thrown.

pubic void read (BufferedReader in) **throws** EOFException, FileNotFoundException
{
  // a checked exception is thrown here !

}

- When the header of method X throws an exception, if an exception occurs, the method terminates. Any method calling the method X must also declare the exception using a throws clause or catch the exception
- How long does an exception last ?
  - Until it is caught.
  - If not caught at the current method, VM looks for the corresponding catch block in the caller of the method.
  - If still not caught (even in the main), Java displays the message. See the figure in the next slide.



6.8.4 Defining Exception Classes

- By convention, most new exception types extend Exception,a subclass of Throwable.
- Exceptions are primarily checked exceptions. A unchecked exception class should extend RuntimeException .
- Should provide two constructors: a default constructor and a constructor with a string parameter that signifies the reason for the exception.

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

- The above code defines a checked exception class which inherits from the Exception class but not from RuntimeException.

6.9 When Not to Use Inheritance

- Use inheritance to model "is-a" relationship.
- Use aggregation (instance fields) for has-a relationship.

Example 1

- From a C++ tutorial

```
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;
}
```

Remedy: Use aggregation

```
public class Circle // OK
{ public Circle(Point center, int radius) { ... }
  public void draw(Graphics g) { ... }
  public void translate(int dx, int dy) { ... }

  private Point center; // A Circle has a center point.
  private int radius;
}
```

Example 2

- java.util.Stack in the Java standard library extends Vector.

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

- Considering the Stack class inherits all Vector methods, the above design choice is a bad idea because it violates the principle of data abstraction. (The remove and add operations are done on the top element only.) However, this version of stacks can insert/remove in the middle of the stack using the inherited Vector methods.
- Remedy: Use aggregation

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