Chapter 13
Object-Oriented Programming
Objectives

• Design and build class hierarchies
• Understand inheritance and polymorphism
• Override inherited methods
• Build abstract classes
Object-Oriented Programming

- OOP: using class hierarchies to model object relations
- Review concepts
  - A subclass is a specialized version of a superclass
  - Subclasses inherit the members of their superclasses
- A class hierarchy familiar to biologists
  - Kingdom, phylum, class, order, family, genus, species
- OO programmers should think like biologists
Object-Oriented Programming (continued)

FIGURE 13-1 Biological hierarchical classification
Introductory Example: Drawing Shapes

- Problem: create logo using square, circle, triangle
Introductory Example: Drawing Shapes (continued)

FIGURE 13-2  A logo sketch
Design

• The user story for Logo.java
  – Open a window
  – Draw a red square, a green circle, and a blue triangle

• Modeling the objects in the program
  – Use the Plotter class to create the window
  – Build Square, Circle, and Triangle classes

• Members common to the three new classes
  – public getColor()
  – public draw()
  – private Color myColor
Design (continued)

<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
<th>Method</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Square</td>
<td>public Square(Color, ...) { ... }</td>
<td>public Color getColor() { ... }</td>
<td>public void draw() { ... }</td>
</tr>
<tr>
<td></td>
<td>public Color getColor() { ... }</td>
<td>public Color getColor() { ... }</td>
<td>private Color myColor;</td>
</tr>
<tr>
<td></td>
<td>public void draw() { ... }</td>
<td>public void draw() { ... }</td>
<td></td>
</tr>
<tr>
<td></td>
<td>private Color myColor;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Circle</td>
<td>public Circle(Color, ...) { ... }</td>
<td>public Color getColor() { ... }</td>
<td>public void draw() { ... }</td>
</tr>
<tr>
<td></td>
<td>public Color getColor() { ... }</td>
<td>public Color getColor() { ... }</td>
<td>private Color myColor;</td>
</tr>
<tr>
<td></td>
<td>public void draw() { ... }</td>
<td>public void draw() { ... }</td>
<td></td>
</tr>
<tr>
<td></td>
<td>private Color myColor;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Triangle</td>
<td>public Triangle(Color, ...) { ... }</td>
<td>public Color getColor() { ... }</td>
<td>public void draw() { ... }</td>
</tr>
<tr>
<td></td>
<td>public Color getColor() { ... }</td>
<td>public Color getColor() { ... }</td>
<td>private Color myColor;</td>
</tr>
<tr>
<td></td>
<td>public void draw() { ... }</td>
<td>public void draw() { ... }</td>
<td></td>
</tr>
<tr>
<td></td>
<td>private Color myColor;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FIGURE 13-3**  **Square, Circle, and Triangle class sketches**
Design (continued)

- Common class members should be in a superclass
  - Principle: define member once, use multiple times
  - Application: design \texttt{Shape} to contain three members
- Extending the class hierarchy
  - A triangle is a polygon and a square is a polygon
  - Create \texttt{Polygon} class to capture common members
- Object-oriented design (OOD)
  - Organize problem classes into an inheritance hierarchy
- Object-oriented programming
  - Process of converting OO design into an OO program
Design (continued)

FIGURE 13-4  A hierarchy of shapes
Implementing the Design

- The **Shape** class is implemented first (most general)
- Members of **Shape**
  - A **Shape()** constructor
  - A instance variable to store the **Color**
  - An accessor for the **Color** variable
  - A **draw()** method
- **Shape** handle refers to Circle, Triangle, Square
- Abstract method: declared (not defined) in superclass
- Abstract class: contains abstract (undefined) methods
Implementing the Design (continued)

```java
/** Shape.java declares an abstract superclass for shapes...
 */
import java.awt.Color;

abstract public class Shape {
    public Shape(Color aColor) { myColor = aColor; }

    public Color getColor() { return myColor; }

    abstract public void draw(Plotter plot);

    private Color myColor;
}
```

FIGURE 13-5 The `Shape` superclass
Implementing the Design (continued)

FIGURE 13-6 Shape handles for subclass objects
Implementing the Design (continued)

• **Circle** extends (becomes a subclass of) **Shape**
• Features of the **Circle** class
  – Instance variables: radius, (x, y) coordinates of center
  – Constructor uses **super()** to call superclass constructor
  – Constructor also initializes **Circle**’s instance variables
  – An accessor for the radius
  – An accessor for a **Point**
  – A definition for **draw()**
• **Point** class: structured type models (x,y) coordinates
• Subclass uses **super** to access superclass members
/** Circle.java models a circle using its (x,y) center and radius... */

import java.awt.Color;

public class Circle extends Shape {
    public Circle(Color aColor, double centerX, double centerY, double radius) {
        super(aColor);
        myX = centerX; myY = centerY;
        myRadius = radius;
    }

    public double getRadius() { return myRadius; }
    public Point getCenter() { return new Point(myX, myY); }

    public void draw(Plotter plot) {
        double diameter = myRadius*2;
        plot.setPenColor( super.getColor() );
        plot.drawOval(myX, myY, diameter, diameter);
    }

    private double myRadius, myX, myY;
}
Implementing the Design (continued)

• Strategy for modeling a polygon
  – Pass the number of corners in the polygon to Polygon
  – Allocate a Point array large enough to store corners

• Features to highlight in the Polygon class
  – Constructor sets the size of a Point array
  – Mutator sets the coordinates of a single Point
  – Accessor returns the coordinate pair of a Point
  – draw() sets the Plotter’s pen color
  – draw() iterates through Point array and draws figure

• protected: restricts superclass access to subclass
Implementing the Design (continued)

```java
/** Polygon.java models a closed polygon...
 */
import java.awt.Color;

public class Polygon extends Shape {
    public Polygon(Color aColor, int numCorners) {
        super(aColor);
        if (numCorners < 3) {
            throw new IllegalArgumentException("Polygon(): "
                + "numCorners must be >= 3");
        }
        myPoints = new Point[numCorners];
    }

    protected void setPoint(int i, double x, double y) {
        myPoints[i] = new Point(x, y);
    }

    public Point getPoint(int i) { return myPoints[i]; }

    public void draw(Plotter plot) {
        plot.setPenColor( super.getColor() );
        Point p0 = myPoints[0], p1 = null;
        for (int i = 1; i < myPoints.length; i++) {
            p1 = myPoints[i];
            plot.drawLine( p0.getX(), p0.getY(),
                p1.getX(), p1.getY() );
            p0 = p1;
        }
        plot.drawLine( p1.getX(), p1.getY(),
            myPoints[0].getX(), myPoints[0].getY() );
    }
```
Implementing the Design (continued)

```java
34  private Point [] myPoints = null;
35  }
```

**FIGURE 13-9  Polygon.java**
Implementing the Design (continued)

- **Strategy for creating a triangle shape**
  - Specifying the color of the triangle
  - Specify the (x,y) coordinates of its three vertices

- **Key features of the Triangle class**
  - Extends the Polygon superclass
  - Constructor sets the vertices of a Triangle

- **Strategy for modeling a Square**
  - Use the color, center coordinates, and side length

- **Square class extends the Polygon superclass**
  - Constructor evaluates and sets the four vertices
/** Triangle.java models a triangle shape,  
 * using the (x,y) coordinates of its 3 vertices...  
 */  
import java.awt.Color;  
  
public class Triangle extends Polygon {  
    public Triangle(Color aColor, double x1, double y1,  
                    double x2, double y2, double x3, double y3) {  
        super(aColor, 3);  
        super.setPoint(0, x1, y1); // corner 1  
        super.setPoint(1, x2, y2); // corner 2  
        super.setPoint(2, x3, y3); // corner 3  
    }  
}  

FIGURE 13-10  Triangle.java
/** Square.java models a square shape,
  * using its (x,y) center and the length of one side...
  */
import java.awt.Color;

public class Square extends Polygon {
    public Square(Color aColor, double centerX, double centerY, 
                   double sideLength) {
        super(aColor, 4);
        double halfSideLength = sideLength / 2;
        super.setPoint(0, centerX - halfSideLength, centerY - halfSideLength); // x of corner 0
        super.setPoint(1, centerX - halfSideLength, centerY + halfSideLength); // y of corner 0
        super.setPoint(2, centerX + halfSideLength, centerY + halfSideLength); // x of corner 1
        super.setPoint(3, centerX + halfSideLength, centerY - halfSideLength); // y of corner 1
    }
}
Implementing the Design (continued)

• **Logo1**: one way to solve the drawing problem

• Key features of the `main()` method in **Logo1**
  – A **Plotter** object is instantiated
  – The axes of **Plotter** are hidden
  – Each of the **Shapes** is instantiated and drawn
    • A specific **Shape** is drawn against a **Plotter** object

• Understanding the use of the **Shape** handle in **Logo1**
  – Refers to three different objects
  – `draw()` method invoked against **Shape** is polymorphic
    • A polymorphic statement has multiple definitions
/** Logol.java draws a logo...
 */
import java.awt.Color;

public class Logol {
    public static void main(String[] args) {
        Plotter plotter = new Plotter(0, 0, 10, 10);
        plotter.hideAxes();
        Shape shape = new Square(Color.RED, 5, 5, 10);
        shape.draw(plotter);
        shape = new Circle(Color.GREEN, 5, 5, 5);
        shape.draw(plotter);
        shape = new Triangle(Color.BLUE, 1, 2, 5, 10, 9, 2);
        shape.draw(plotter);
    }
}
Implementing the Design (continued)

Result:

![Graph showing the result of Logo1.java](image)

**Figure 13-12** Logo1.java
Implementing the Design (continued)

**FIGURE 13-14** A Shape handle refers to a circle object
Implementing the Design (continued)

- **Logo**: more efficient solution to the drawing problem
- Key features of the **Logo** class
  - LinkedList of **Shapes** called **myShapes** is declared
  - The constructor allocates memory to **myShapes**
  - The constructor adds three **Shapes** to **myShapes**
  - **draw()** traverses **myShapes** and draws each **Shape**
  - **main()** calls **Logo** constructor and **draw()**
- The second implementation is more easily reused
  - The ability to specify a **Plotter** adds flexibility
  - New **Shapes** can be added with just a single statement
/** Logo.java draws a logo using a LinkedList...
 */
import java.awt.Color;
import java.util.LinkedList;

public class Logo {
    public Logo() {
        myShapes = new LinkedList<Shape>();
        myShapes.add( new Square(Color.RED, 5, 5, 10) );
        myShapes.add( new Circle(Color.GREEN, 5, 5, 5) );
        myShapes.add( new Triangle(Color.BLUE, 1, 2, 5, 10, 9, 2) );
    }

    public void draw(Plotter plotter) {
        plotter.hideAxes();
        for (Shape shape : myShapes) {
            shape.draw(plotter);
        }
    }

    private LinkedList<Shape> myShapes = null;
}

public static void main(String[] args) {
    continued
Implementing the Design (continued)

```java
24      Logo self = new Logo();
25      self.draw( new Plotter(0, 0, 10, 10) );
26     }
27    }
```

**FIGURE 13-16** Logo.java
Class Hierarchies and Object-Oriented Programming

• Object-oriented design
  – Design classes needed to solve a problem
  – Identify common attributes and/or behaviors
  – Store common attributes/behaviors in superclasses
  – Use the “is a” relationship to organize class hierarchy

• Object-oriented programming
  – Implementing classes conceived during design phase
  – Some features supported by an OOP language
    • Subclasses, superclasses, inheritance, abstract methods
Java’s Class Hierarchy

• **Object class**
  – Provides general-purpose methods; e.g. `toString()`
  – Superclass of every standard Java class; e.g., `Math`
  – Superclass of every nonstandard Java class; e.g., `Logo`
• Every Java class is ultimately a subclass of **Object**
• Class structure diagram: relates program classes
Java’s Class Hierarchy (continued)

FIGURE 13-17  The class hierarchy for drawing logos
The *is a* and *has a* Relationships

- A subclass "is a" special kind of its superclass
- The "is a" relationship applies to all superclasses
  - Ex: *Triangle is a Polygon, a Shape, an Object*
- The "has a" relationship
  - A relationship of container to an object contained
  - Ex. *Logo class has a LinkedList<Shape>*
- How class structure diagram represents relationships
  - "is a": arrows point from a subclass to superclass
  - "has a": diamond-headed line between classes
The *is a* and *has a* Relationships (continued)

![Class structure diagram for Logo.java](image)

**FIGURE 13-18** Class structure diagram for Logo.java
The *is a* and *has a* Relationships (continued)

- Use “is a” to begin building class structure diagram
  - If class X is a particular kind of class Y, X can extend Y
- Illustration: building a **Tree** class structure diagram
  - Apple tree is a fruit tree; so is peach tree and pear tree
  - Chestnut tree is a nut tree; walnut tree is a nut tree
  - A fruit tree is a tree
  - A nut tree is a tree
- Two forms of principle governing “is a” relationship
  - Superclasses are generalizations of their subclasses
  - Subclasses are specializations of their superclasses
The *is a* and *has a* Relationships (continued)

**FIGURE 13-19** A Tree class structure diagram
The *is a* and *has a* Relationships
(continued)

• How to determine if a “has a” relationship is needed
  – Consider defining class X as a subclass of class Y
  – Ensure that every message to Y object can go to X
  – If some messages to X are inappropriate, X “has a” Y

• Illustrating the development of a “has a” relationship
  – Attempt to define `Orchard` as an `ArrayList<Tree>`
  – Recognize inappropriate messages; e.g., `clone()`
  – Redesign relationship so `Orchard` is not a subclass
  – Make `ArrayList<Tree>` an attribute of `Orchard`
The *is a* and *has a* Relationships (continued)

FIGURE 13-20 An **orchard** class structure diagram
Hierarchy in the Java API

- All Java standard classes are part of Java API
- Java class relations can be traced up to Object
- Illustration: tracing the Double wrapper class
  - Double is a subclass of Number
  - Number is a subclass of Object
- How to view the API for a superclass
  - Click on the name of the superclass
  - Example: click java.lang.Number for Number
- The API for a class also lists that class’ subclasses
Hierarchy in the Java API (continued)

FIGURE 13-22  The subclasses of class `Number`
Bottom-Up Design versus Top-Down Programming

• Object-oriented design is a bottom-up process
  – Design the most specific classes first (from user story)
  – Identify common attributes/behaviors using “is a”
  – Continue upward design toward greater generalization

• Object-oriented programming is a top-down process
  – Build and test the topmost superclass
  – Build and test each of the top superclass’ subclasses

• Rationale for OOP movement from top to bottom
  – Subclass cannot be compiled until superclass defined

• Build program to solve problem after testing classes
Example 2: Object-Oriented T-Shirt Sales

- **User story for TShirtOrder**
  - 1. Display a menu listing t-shirt sizes along with prices
  - 2. Read the user’s choice
  - 3. Display that t-shirt and its price
  - 4. Add that t-shirt’s price to the total
  - 5. Repeat steps 1-4 as long as the user is not done

- **Nouns and noun phrases identified in user story**
  - Program, menu, various t-shirts, price, t-shirt list, total

- **T-shirt’s will be related in a class hierarchy**

- **Key verbs: display, read, get, add, repeat, list**
Example 2: Object-Oriented T-Shirt Sales (continued)

FIGURE 13-23  A TShirtOrder class hierarchy
Example 2: Object-Oriented T-Shirt Sales (continued)

<table>
<thead>
<tr>
<th>Verb Phrase</th>
<th>Method or Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>display ...</td>
<td><code>System.out.print()</code> or <code>println()</code></td>
</tr>
<tr>
<td>read the user’s choice</td>
<td><code>Scanner#nextInt()</code></td>
</tr>
<tr>
<td>get a t-shirt’s price</td>
<td><code>TShirt#getPrice()</code></td>
</tr>
<tr>
<td>add the t-shirt’s price</td>
<td><code>+=</code></td>
</tr>
<tr>
<td>repeat steps ...</td>
<td><code>do</code> <code>loop</code></td>
</tr>
<tr>
<td>list order’s t-shirts</td>
<td><code>TShirtOrder#printShirts()</code>, for each <code>loop</code></td>
</tr>
<tr>
<td>list order’s total</td>
<td><code>println()</code>, <code>TShirtOrder#getTotal()</code></td>
</tr>
</tbody>
</table>
Example 2: Object-Oriented T-Shirt Sales (continued)

- Members of the `TShirt` superclass
  - String type instance variable for the t-shirt size
  - Constructor to set the size of the t-shirt
  - An accessor to return the size of the t-shirt
  - An accessor to return a default price of a t-shirt
  - A `toString()` method

- Object-based programming does not use inheritance

- `TShirt` in Figure 10-7 vs `TShirt` in Figure 13-24
  - First `TShirt` is built under object-based paradigm
  - Second `TShirt` is object-oriented (uses inheritance)
Example 2: Object-Oriented T-Shirt Sales (continued)

```java
/** TShirt.java is the superclass for different sized t-shirts... */
public class TShirt {
    public TShirt(String size) {
        if (!size.equals("XS") || size.equals("S") || size.equals("M")
            || size.equals("L") || size.equals("XL")) {
            throw new IllegalArgumentException("TShirt(): bad size: "
                + size);
        }
        mySize = size;
    }

    public String getSize() { return mySize; }
    public double getPrice() { return 10.00; }
    public String toString() { return mySize + " TShirt"; }

    private String mySize;
}
```

**Figure 13-24 TShirt.java**
Example 2: Object-Oriented T-Shirt Sales (continued)

- The members of the XSmallTShirt subclass
  - A constructor that only calls the TShirt constructor
  - A method to return an XSmallTShirt specific price
- Distinguish method overriding from overloading
- Method overriding
  - A subclass method redefining an inherited method
- Method overloading
  - Methods defining multiple versions of one operation
- `getPrice()` overrides method inherited from TShirt
Example 2: Object-Oriented T-Shirt Sales (continued)

```java
/** XSmallTShirt.java models an extra-small t-shirt...
 */

public class XSmallTShirt extends TShirt {
    public XSmallTShirt()    { super("XS"); }
    public double getPrice() { return 9.00; }
}
```

FIGURE 13.25  XSmallTShirt.java
Example 2: Object-Oriented T-Shirt Sales (continued)

- The members of the SmallTShirt subclass
  - A constructor that only calls the TShirt constructor
  - Uses the getPrice() method inherited from TShirt

- MediumTShirt and LargeTShirt classes
  - Structures are similar to that of SmallTShirt

- The members of the XLargeTShirt subclass
  - A constructor that only calls the TShirt constructor
  - getPrice() method that overrides the inherited method

- TShirtOrder uses TShirt classes to solve problem
Example 3: Object-Oriented Business Software

• The basic user story
  – Read the payrollData.txt file
  – Compute and display employee pay

• Instructions for paying an employee
  – A salaried employee should be paid her salary
  – Hourly employee (no OT) is paid wage * hours worked
  – Hourly employee (with OT) is paid regular plus bonus

• Noun phrases indicate four classes should be built
  – Employee is superclass of salaried and hourly types
  – A Payroll has an Employee
Example 3: Object-Oriented Business Software (continued)

FIGURE 13-31  Class hierarchy for Amazons.com payroll program (2)
Example 3: Object-Oriented Business Software (continued)

- Members of the **Employee** class
  - Instance variables for name and employee ID
  - A default constructor to initialize instance variables
  - A parameterized constructor using input values
  - Two accessors to read the instance variables
  - `toString()` to show `String` values of data members
  - Method to read the name and ID from keyboard
  - Abstract method to return the pay value
Example 3: Object-Oriented Business Software (continued)

```java
/** Employee.java provides a simple model of an employee...
 */
import java.util.Scanner;

abstract public class Employee {
   public Employee() {
      myName = "";
      myID = 0;
   }

   public Employee(String name, int id) {
      myName = name;
      myID = id;
   }

   public String getName() { return myName; }
   public int getID() { return myID; }
   public String toString() { return myName + "\t" + myID; }

   abstract double getPay();

   public void read(Scanner in) {
      myName = in.next();
      myID = in.nextInt();
   }

   private String myName = null;
   private int myID;
}
```

FIGURE 13-32 Employee.java
Example 3: Object-Oriented Business Software (continued)

- Some observations of `SalariedEmployee` subclass
  - Call to `super()` made by both constructors
  - `toString()` uses `super.toString()`
  - `read()` method uses `super.read(in)`
  - Do not omit `super` in a message to a superclass
    - If method names are the same, call becomes recursive
      - `getPay()` is defined so the class can be instantiated
- `HourlyEmployee` is similar to `SalariedEmployee`
  - Exception: `getPay()` has a class specific definition
- All three `Employee` classes should be tested
Example 3: Object-Oriented Business Software (continued)

```java
/** SalariedEmployee.java models a salaried employee...
 */
import java.util.Scanner;

public class SalariedEmployee extends Employee {
    public SalariedEmployee() {
        super();
        mySalary = 0.0;
    }

    public SalariedEmployee(String name, int id, double salary) {
        super(name, id);
        mySalary = salary;
    }

    public double getSalary() { return mySalary; }

    public String toString() {
        return super.toString() + "\t" + mySalary;
    }

    public void read(Scanner in) {
        super.read(in);
        mySalary = in.nextDouble();
    }

    public double getPay() { return mySalary; }

    private double mySalary;
}
```

Figure 13-33  SalariedEmployee.java
Example 3: Object-Oriented Business Software (continued)

- Algorithm used by Payroll to solve the problem
  - 1. Wrap a Scanner named fin around payrollData.txt
  - 2. While fin.hasNext():
    - Read employee type string from fin into employeeKind
    - Use employeeKind to construct an Employee subtype
    - Fill employee with data by sending it read(fin)
    - Send getName(); getPay() to employee and display
  - 3. Close fin

- try-catch-finally is used to handle exceptions
- Class class: used to create a subtype of Employee
Example 3: Object-Oriented Business Software (continued)

```java
/** Payroll.java computes the payroll for Amazons.com... */
import java.util.Scanner;
import java.io.*;  // File, ...

public class Payroll {
    public void run() {
        Scanner fin = null;
        try {
            fin = new Scanner( new File("payrollData.txt") );
            Employee employee = null;
            while ( fin.hasNext() ) {
                String employeeKind = fin.next() + "Employee";
                employee = Class.forName(employeeKind)
                    .asSubclass(Employee.class)
                    .newInstance();
                employee.read(fin);
                System.out.printf("%20s: $%7.2f\n", employee.getName(),
                            employee.getPay());
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            fin.close();
        }
    }

    public static void main(String[] args) {
        Payroll self = new Payroll();
        self.run();
    }
}
```

Result:

<table>
<thead>
<tr>
<th>Name</th>
<th>Payroll</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ortrera</td>
<td>$1000.00</td>
</tr>
<tr>
<td>Orithya</td>
<td>$840.00</td>
</tr>
<tr>
<td>Antiope</td>
<td>$850.00</td>
</tr>
<tr>
<td>Hippolyte</td>
<td>$800.00</td>
</tr>
<tr>
<td>Penethesle</td>
<td>$900.00</td>
</tr>
<tr>
<td>Clonie</td>
<td>$400.00</td>
</tr>
<tr>
<td>Holpadias</td>
<td>$300.00</td>
</tr>
<tr>
<td>Thalestris</td>
<td>$750.00</td>
</tr>
</tbody>
</table>
Example 3: Object-Oriented Business Software (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Melanippe</td>
<td>$600.00</td>
</tr>
<tr>
<td>Ainia</td>
<td>$900.00</td>
</tr>
<tr>
<td>Antibrote</td>
<td>$700.00</td>
</tr>
<tr>
<td>Helene</td>
<td>$200.00</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

**FIGURE 13-35** Payroll.java
Summary

- Object-oriented design: bottom-up process for organizing a problem’s classes into a hierarchy
- Object-oriented programming: top-down process for implementing an object-oriented design
- Abstract method: a class declared in a superclass and defined in a subclass
- Abstract class: a class that contains an abstract method
- A subclass can send a message to its superclass by using the `super` notation
Summary (continued)

• Access specifiers: `private`, `protected`, `public`
• In Java, `Object` is the superclass of all classes
• Classes are associated by the “is a” and “has a” relationships
• Mutator (setter): method used to change the value of an instance variable
• Polymorphic method: a method that can be invoked against multiple types