Polymorphism
(Deitel chapter 10)
(Old versions: chapter 9)

Plan

• Introduction
• Relationships Among Objects in an Inheritance Hierarchy
• Polymorphism Examples
• Abstract Classes and Methods
• Example: Inheriting Interface and Implementation
• final Methods and Classes
• Example: Payroll System Using Polymorphism
• Example: Creating and Using Interfaces
• Nested Classes
• Type-Wrapper Classes for Primitive Types

Introduction

• Polymorphism
  - “Program in the general”
  - Treat objects in same class hierarchy as superclass objects
  - Abstract class
    • Common functionality
    • Makes programs extensible
      • New classes added easily, can still be processed

Introduction

• Treat object not as the specific type, but instead as the base type
• Allows to write code that doesn’t depend on specific types
• In the shape example, methods manipulate generic shapes without respect to whether they’re circles, squares, etc.

Introduction

A method in Java

```java
void doStuff(Shape s) {
  s.erase();
  // ...
  s.draw();
}
```

This method speaks to any `Shape`, so it is independent of the specific type of object

```java
Circle c = new Circle();
Triangle t = new Triangle();
Line l = new Line();
doStuff(c);
doStuff(t);
doStuff(l);
```

calls to `doStuff()` automatically work correctly, regardless of the exact type of the object.

d`oStuff(Shape s)` is being passed into a method that’s expecting a `Shape`.

Introduction

```
Shape
  + Circle
  + Square
  + Triangle
```

```
BirdController
  + move
  + fly
  + eat

Bird
  + move
  + fly
  + eat

Genus
  + move

Penguin
  + move
  + fly
```

• Treat object not as the specific type, but instead as the base type
• Allows to write code that doesn’t depend on specific types
• In the shape example, methods manipulate generic shapes without respect to whether they’re circles, squares, etc.

Notice that it doesn’t say “If you’re a Circle, do this, if you’re a Square, do that, etc.”

Often, you don’t want to create an object of the base class, only to upcast to it (abstract keyword).
Introduction

• One way to determine object’s class
  – Give base class an attribute
    • shapeType in class Shape
  – Use switch to call proper print function

• Many problems
  – May forget to test for case in switch
  – If add/remove a class, must update switch structures
• Time consuming and error prone

• Better to use polymorphism
  – Less branching logic, simpler programs, less debugging

• In our examples
  – Use abstract superclass Shape
    • Defines common interface (functionality)
      • Point, Circle and Cylinder inherit from Shape
        Each class may have its own draw method (quite different)

• If we use a super class reference to refer to a subclass object and invoke the draw method, the program choose the correct draw method dynamically.
  • This is called dynamic method binding

Relationships Among Objects in an Inheritance Hierarchy

• Previously
  – Circle inherited from Point
  – Manipulated Point and Circle objects using references to invoke methods

• Here
  – Invoking superclass methods from subclass objects
  – Using superclass references with subclass-type variables

• Key concept
  – subclass object can be treated as superclass object
    • “is-a” relationship
    • superclass is not a subclass object

Invoking Superclass Methods from Subclass Objects

• Store references to superclass and subclass objects
  – Assign a superclass reference to superclass-type variable
  – Assign a subclass reference to a subclass-type variable
    • Both straightforward
  – Assign a subclass reference to a superclass variable
    • “is-a” relationship

Outline

• In Fig. 10.1: HierarchyRelationshipTest1.java
  – Assigning superclasses and subclasses references to superclass- and subclass-type variables

public class HierarchyRelationshipTest1 {
  
  public static void main( String[] args ) {
    // assign superclass reference to superclass-type variable
    Point3 point = new Point3( 30, 50 );
    
    // assign subclass reference to subclass-type variable
    Circle4 circle = new Circle4( 120, 89, 2.7 );
    
    // invoke toString on superclass object using superclass variable
    String output = "Call Point3’s toString with superclass reference to superclass object: 
" + point.toString();
    
    // invoke toString on subclass object using subclass variable
    output += "Call Circle4’s toString with subclass reference to subclass object: 
" + circle.toString();
    
    // invoke toString on subclass object using superclass variable
    Point3 pointRef = circle;
    output += "Call Circle4’s toString with superclass reference to subclass object: 
" + pointRef.toString();
    
    JOptionPane.showMessageDialog( null, output );  // display output
  
  } // end main

} // end class HierarchyRelationshipTest1
Using Superclass References with Subclass-Type Variables

- Previous example
  - Assigned subclass reference to superclass-type variable
    - Circle "is a" Point
- Assign superclass reference to subclass-type variable
  - Compiler error
    - No "is a" relationship
    - Point is not a Circle
    - Circle has data/methods that Point does not
      - setRadius (declared in Circle) not declared in Point
  - Cast superclass references to subclass references
    - Called downcasting
    - Invoke subclass functionality

Polymorphism Examples

- Examples
  - Suppose Rectangle derives from Quadrilateral
    - Rectangle more specific than Quadrilateral
    - Any operation on Quadrilateral can be done on Rectangle (i.e., perimeter, area)
  - Suppose designing video game
    - Superclass SpaceObject
      - Subclasses Martian, SpaceShip, LaserBeam
      - Contains method draw
        - To refresh screen
        - Send draw message to each object
        - Same message has "many forms" of results

Abstract Classes and Methods

- Abstract classes
  - Are superclasses (called abstract superclasses)
  - No objects cannot be instantiated
  - Incomplete
    - subclasses fill in "missing pieces"
- Concrete classes
  - Can be instantiated
  - Implement every method they declare
  - Provide specifics
- Example
  - Abstract superclass TwoDimensionalObject
  - Derive concrete classes Square, Circle, Triangle

Abstract Classes and Methods (Cont.)

- Abstract classes not required, but reduce client code dependencies
  - To make a class abstract
    - Declare with keyword abstract
    - Contain one or more abstract methods
      - public abstract void draw();
  - Abstract methods
    - No implementation, must be overridden
Abstract Classes and Methods (Cont.)

- Application example
  - Abstract class Shape
    - Declares draw as abstract method
    - Circle, Triangle, Rectangle extends Shape
      - Each must implement draw
      - Each object can draw itself

Example: Inheriting Interface and Implementation

- Make abstract superclass Shape
  - Abstract method (must be implemented)
    - getName, print
  - Default implementation does not make sense
    - Methods may be overridden
      - getArea, getVolume
        - Default implementations return 0.0
      - If not overridden, uses superclass default implementation
      - Subclasses Point, Circle, Cylinder

Outline

Shape
  - Point
  - Circle
  - Cylinder

Example: Inheriting Interface and Implementation

Polymorphic interface for the Shape hierarchy classes.
final Methods and Classes

- **final methods**
  - Cannot be overridden
  - private methods are implicitly final

- **final classes**
  - Cannot be subclasses, i.e. cannot be extended
  - Methods in final classes are implicitly final
  - e.g., class String

Notes

- Polymorphism causes less branching logic in favor of simpler sequential code.
- Facilitates testing, debugging, maintenance.
- With polymorphism the programmer can deal in generalities and let the run-time environment deal with the specifics.
- Polymorphism promotes extensibility

- **final methods** cannot be overridden in subclasses. If sent to subclasses will be responded identically rather than polymorphically.
- Compiler can decide to inline final methods calls to improve performance.
Example: Payroll System Using Polymorphism

- Create a payroll program
  - Use abstract methods and polymorphism
- Problem statement
  - 4 types of employees, paid weekly
    - Salaried (fixed salary, no matter the hours)
    - Hourly (overtime [>40 hours] pays time and a half)
    - Commission (paid percentage of sales)
      - Boss wants to raise pay by 10%

Example: Payroll System Using Polymorphism

- Superclass Employee
  - Abstract method earnings (returns pay)
    - abstract because need to know employee type
    - Cannot calculate for generic employee
  - Other classes extend Employee
    - SalariedEmployee
    - HourlyEmployee
    - CommissionEmployee
    - BasePlusCommissionEmployee

Outline

Employee.java

```
1 // Fig. 10.12: Employee.java
2 // Employee abstract superclass.
3
4 public abstract class Employee {
5   private String firstName;
6   private String lastName;
7   private String socialSecurityNumber;
8
9   // constructor
10  public Employee( String first, String last, String ssn ) {
11     firstName = first;
12     lastName = last;
13     socialSecurityNumber = ssn;
14   }
15
16   // set first name
17   public void setFirstName( String first ) {
18     firstName = first;
19   }
20
21   // return first name
22   public String getFirstName() {
23     return firstName;
24   }
25
26   // set last name
27   public void setLastName( String last ) {
28     lastName = last;
29   }
30
31   // return last name
32   public String getLastName() {
33     return lastName;
34   }
35
36   // set social security number
37   public void setSocialSecurityNumber( String number ) {
38     socialSecurityNumber = number;  // should validate
39   }
40
41   // return social security number
42   public String getSocialSecurityNumber() {
43     return socialSecurityNumber;
44   }
45
46   // return String representation of Employee object
47   public String toString() {
48     return getFirstName() + " " + getLastName() +
49       " social security number: " + getSocialSecurityNumber();
50   }
51
52   // abstract method overridden by subclasses
53   public abstract double earnings();
54
55 } // end abstract class Employee
```

Outline

SalariedEmployee.java

```
1 // Fig. 10.13: SalariedEmployee.java
2 // SalariedEmployee class extends Employee.
3
4 public class SalariedEmployee extends Employee {
5   private double weeklySalary;
6
7   // constructor
8   public SalariedEmployee( String first, String last,
9     String socialSecurityNumber, double salary ) {
10     super( first, last, socialSecurityNumber );
11     setWeeklySalary( salary );
12   }
13
14   // set salaried employee's salary
15   public void setWeeklySalary( double salary ) {
16     weeklySalary = salary < 0.0 ? 0.0 : salary;
17   }
18
19   // return salaried employee's salary
20   public double getWeeklySalary() {
21     return weeklySalary;
22   }
23
24 } // end abstract class SalariedEmployee
```

Outline

Employee.java

```
1 // Fig. 10.14: Employee.java
2 // Abstract method earnings overridden by subclasses.
3
4 public abstract double earnings();
5
6 // return String representation of Employee object
7 public String toString() {
8   return super.toString();
9     + " earnings: " + earnings();
10 }
11
12 } // end abstract class Employee
```

Outline

SalariedEmployee.java

```
1 // Fig. 10.15: SalariedEmployee.java
2 // SalariedEmployee class extends Employee.
3
4 public class SalariedEmployee extends Employee {
5   private double weeklySalary;
6
7   // constructor
8   public SalariedEmployee( String first, String last,
9     String socialSecurityNumber, double salary ) {
10     super( first, last, socialSecurityNumber );
11     weeklySalary = salary;
12   }
13
14   // return salaried employee's salary
15   public double getWeeklySalary() {
16     return weeklySalary;
17   }
18
19 } // end abstract class SalariedEmployee
```
```java
public double earnings() {
    // override abstract method earnings in Employee
    // calculate hourly employee's pay;
    return wage * hours;
}
```
Example: Creating and Using Interfaces

- Use interface Shape
  - Replace abstract class Shape
- Interface
  - Declaration begins with interface keyword
  - Classes implement an interface (and its methods)
  - Contains public abstract methods
    - Classes (that implement the interface) must implement these methods
    - Implementing a method is like signing a contract
    - If a class leaves a method in the interface undefined, the class must be declared abstract
// Fig. 10.19: Point.java
// Point class declaration implements interface Shape.
public class Point extends Object implements Shape {

private int x;  // x part of coordinate pair
private int y;  // y part of coordinate pair

// constructor
public Point( int xValue, int yValue )
{
    // implicit call to Object constructor occurs here
    x = xValue;  // no need for validation
    y = yValue;  // no need for validation
}

// set x in coordinate pair
public void setX( int xValue )
{
    x = xValue;  // no need for validation
}

// return x from coordinate pair
public int getX()
{
    return x;
}

// set y in coordinate pair
public void setY( int yValue )
{
    y = yValue;  // no need for validation
}

// return y from coordinate pair
public int getY()
{
    return y;
}

// declare abstract method getArea
public double getArea()
{
    return 0.0;
}

// declare abstract method getVolume
public double getVolume()
{
    return 0.0;
}

// override abstract method getName to return "Point"
public String getName()
{
    return "Point";
}

// override toString to return String representation of Point
public String toString()
{
    return "[" + getX() + ", " + getY() + "]";
}
}

// Fig. 10.20: InterfaceTest.java
// Test Point, Circle, Cylinder hierarchy with interface Shape.
import java.text.DecimalFormat;
import javax.swing.JOptionPane;

public class InterfaceTest {

    public static void main( String args[] )
    {
        // set floating-point number format
        DecimalFormat twoDigits = new DecimalFormat( "0.00" );

        // create Point, Circle and Cylinder objects
        Point point = new Point( 7, 11 );
        Circle circle = new Circle( 22, 8, 3.5 );
        Cylinder cylinder = new Cylinder( 20, 30, 3.3, 10.75 );

        // obtain name and string representation of each object
        String output = point.getName() + ": " + point + 
                        circle.getName() + ": " + circle + 
                        cylinder.getName() + ": " + cylinder + 
                        "\n";

        // create Shape array
        Shape arrayOfShapes[] = new Shape[ 3 ];  // create Shape array

        // aim arrayOfShapes[ 0 ] at subclass Point object
        arrayOfShapes[ 0 ] = point;

        // aim arrayOfShapes[ 1 ] at subclass Circle object
        arrayOf Shapes[ 1 ] = circle;

        // aim arrayOfShapes[ 2 ] at subclass Cylinder object

        // loop through arrayOfShapes to get name, string representation, area
        // and volume of every shape in array
        for ( int i = 0; i < arrayOfShapes.length; i++ )
            output += "\n\n" + arrayOfShapes[ i ].getName() + ": " + arrayOfShapes[ i ].toString() + 
                        "\nArea = " + twoDigits.format( arrayOfShapes[ i ].getArea() ) + 
                        "\nVolume = " + twoDigits.format( arrayOfShapes[ i ].getVolume() ) + 
                        "\n";

        JOptionPane.showMessageDialog( null, output );  // display output
        System.exit( 0 );
    }
}

// end class InterfaceTest

Type-Wrapper Classes for Primitive Types

• Type-wrapper class
  – Each primitive type has a **Type-wrapper class**
    • `Character`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Boolean`.
  – Enable to manipulate primitive types as objects of class `Object`
    • Therefore, primitive types can be processed polymorphically if maintained as objects of the type wrapper classes.
  – Each of the type wrapper is declared as `final`
    • Methods implicitly `final`, cannot be overridden
  – Numeric classes inherit from class `Number`