Chapter 9 - Object-Oriented Programming

- Object-oriented programming
  - Inheritance
    - Software reusability
    - Classes are created from existing ones
      - Absorbing attributes and behaviors
      - Adding new capabilities
      - **`Class Convertible`** inherits from class **`Automobile`**
      - **`Superclass v.s. subclass`**
      - **`Single inheritance v.s. multiple inheritance`**
  - Polymorphism
    - Enables developers to write programs in general fashion
      - Handle a variety of existing and yet-to-be-specified related classes
    - Helps add new capabilities to system

Introduction

- Object-oriented programming
  - Inheritance
    - *Sub-class* inherits from *super-class*
      - Subclass usually adds their characteristics of instance variables and methods
    - Single vs. multiple inheritance
      - Single inheritance → **class subclass extends superclass**
      - Java does not support multiple inheritance
        - Interfaces (discussed later) achieve the same effect

        Ex. **class subclass implements superclass1, superclass2, …**
    - "**Is a**" relationship (an object of a subclass type may be treated as its super-class type)
  - Composition
    - "**Has a**" relationship(an object has one or more objects of other classes)

Superclasses and Subclasses

- "Is a" Relationship
  - Object "is an" object of another class
    - Rectangle "is a" quadrilateral
      - Class **`Rectangle`** inherits from class **`Quadrilateral`**
  - Form tree-like hierarchical structures
- "Has a" Relationship
  - Object "is composed of" objects of another class
    - Car "has a" steering wheel
      - Class **`car`** is composed of class **`wheel`**

Superclass	Subclasses
`Student`	`GraduateStudent` `UndergraduateStudent`

`Shape`	`Circle`
	`Triangle`
	`Rectangle`

`Loan`	`CarLoan`
	`HomeImprovementLoan`
	`MortgageLoan`

`Employee`	`FacultyMember` `StaffMember`
`Account`	`CheckingAccount` `SavingsAccount`

Fig. 9.1 Some simple inheritance examples in which the subclass "is a" superclass.

Fig. 9.2 An inheritance hierarchy for university `CommunityMembers`.

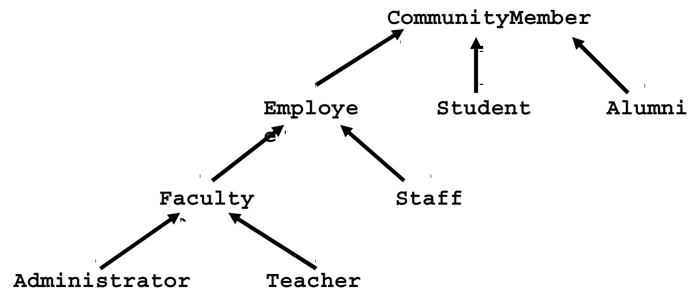
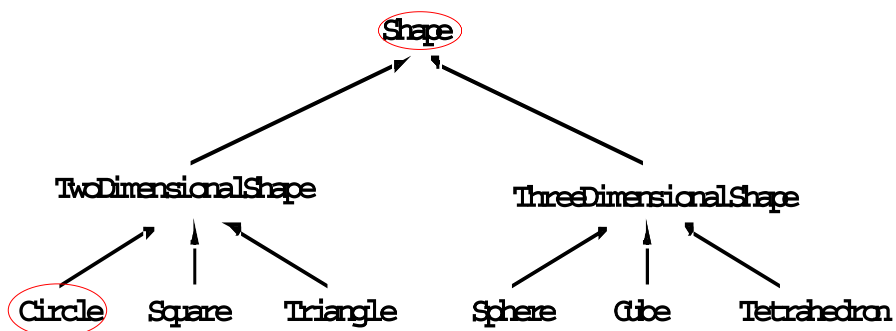


Fig. 9.3 A portion of a `Shape` class hierarchy.



protected Members

- **`protected`** access members
  - Between **`public`** and **`private`** in protection
  - Accessed only by
    - **Superclass methods**
    - **Subclass methods**
    - Methods of classes in same package
      - package access

Relationship between Superclass and Subclass Objects

- Subclass object
  - Can be treated as superclass object
    - Reverse is not true
      - **`Shape`** is not always a **`Circle`**
  - Every class implicitly extends **`java.lang.Object`**
    - Unless specified otherwise in class definition's first line

7

```
1   // Fig. 9.4: Point.java
2   // Definition of class Point
3
4   public class Point {
5      protected int x, y; // coordinates of Point
6
7      // No-argument constructor
8      public Point()
9      {
10        // implicit call to superclass constructor occurs here
11        setPoint( 0, 0 );
12     }
13
14     // constructor
15     public Point( int xCoordinate, int yCoordinate )
16     {
17        // implicit call to superclass constructor occurs here
18        setPoint( xCoordinate, yCoordinate );
19     }
20
21     // set x and y coordinates of Point
22     public void setPoint( int xCoordinate, int yCoordinate )
23     {
24        x = xCoordinate;
25        y = yCoordinate;
26     }
27
28     // get x coordinate
29     public int getX()
30     {
31        return x;
32     }
33
```

Outline

`Point.java`

Line 5
`protected` members prevent clients from direct access (unless clients are **`Point`** subclasses or are in same package)

8

```
34    // get y coordinate
35    public int getY()
36    {
37       return y;
38    }
39
40    // convert into a String representation
41    public String toString()
42    {
43       return "[" + x + ", " + y + "]";
44    }
45
46 }  // end class Point
```

Outline

Point.java

9

```
1  // Fig. 9.5: Circle.java
2  // Definition of class Circle
3
4  public class Circle extends Point {  // inherits from Point
5     protected double radius;
6
7     // no-argument constructor
8     public Circle()
9     {
10       // implicit call to superclass constructor occurs here
11       setRadius( 0 );
12    }
13
14    // constructor
15    public Circle( double circleRadius, int xCoordinate,
16       int yCoordinate )
17    {
18       // call superclass constructor to set coordinates
19       super( xCoordinate, yCoordinate );
20
21       // set radius
22       setRadius( circleRadius );
23    }
24
25    // set radius of Circle
26    public void setRadius( double circleRadius )
27    {
28       radius = ( circleRadius >= 0.0 ? circleRadius : 0.0 );
29    }
30
```

Outline

Circle.java

Line 4
Circle is a **Point** subclass

Line 4
Circle inherits **Point**'s **protected** variables and **public** methods (except for constuctor)

Line 10
Implicit call to **Point** constructor

Line 19
Explicit call to **Point** constructor using **super**

10

```java
31     // get radius of Circle
32     public double getRadius()
33     {
34        return radius;
35     }
36
37     // calculate area of Circle
38     public double area()
39     {
40        return Math.PI * radius * radius;
41     }
42
43     // convert the Circle to a String
44     public String toString()
45     {
46        return "Center = " + "[" + x + ", " + y + "]" +
47               "; Radius = " + radius;
48     }
49
50  }  // end class Circle
```

Outline

`Circle.java`

Lines 44-48
Override method **`toString`** of class **`Point`** by using same signature


```java
1   // Fig. 9.6: InheritanceTest.java
2   // Demonstrating the "is a" relationship
3
4   // Java core packages
5   import java.text.DecimalFormat;
6
7   // Java extension packages
8   import javax.swing.JOptionPane;
9
10  public class InheritanceTest {
11
12     // test classes Point and Circle
13     public static void main( String args[] )
14     {
15        Point point1, point2;
16        Circle circle1, circle2;
17
18        point1 = new Point( 30, 50 );
19        circle1 = new Circle( 2.7, 120, 89 );
20
21        String output = "Point point1: " + point1.toString() +
22           "\nCircle circle1: " + circle1.toString();
23
24        // use "is a" relationship to refer to a Circle
25        // with a Point reference
26        point2 = circle1;   // assigns Circle to a Point reference
27
28        output += "\n\nCircle circle1 (via point2 reference): " +
29           point2.toString();
30
31        // use downcasting (casting a superclass reference to a
32        // subclass data type) to assign point2 to circle2
33        circle2 = ( Circle ) point2;
34
```

Outline

`InheritanceTest.java`

Lines 18-19
Instantiate objects

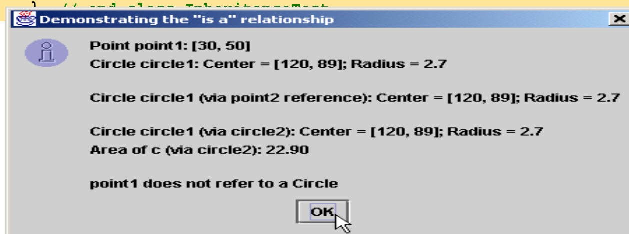
Line 22
`Circle` invokes method **`toString`**

Line 26
Superclass object references subclass

Line 29
`Point` invokes **`Circle`**'s **`toString`** method

Line 33
Downcast **`Point`** to **`Circle`**

```
35          output += "\n\nCircle circle1 (via circle2): " +
36             circle2.toString();
37
38          DecimalFormat precision2 = new DecimalFormat( "0.00" );
39          output += "\nArea of c (via circle2): " +
40             precision2.format( circle2.area() );
41
42          // attempt to refer to Point object with Circle reference
43          if ( point1 instanceof Circle ) {
44             circle2 = ( Circle ) point1;
45             output += "\n\ncast successful";
46          }
47          else
48             output += "\n\npoint1 does not refer to a Circle";
49
50          JOptionPane.showMessageDialog( null, output,
51             "Demonstrating the \"is a\" relationship",
52             JOptionPane.INFORMATION_MESSAGE );
53
54          System.exit( 0 );
55       }
56
57    }
```



Outline

InheritanceTest.java

Line 36
Circle invokes its overridden **toString** method

Line 40
Circle invokes method **area**

Line 43
Use **instanceof** to determine if **Point** refers to **Circle**

Line 44
If **Point** refers to **Circle**, cast **Point** as **Circle**

13

Constructors and Finalizers in Subclasses

- Superclass constructor
  - Initializes superclass instance variables of subclass
  - Not inherited by subclass
  - Called by subclass
    - Implicitly or explicitly with **super** reference
- **finalize** method
  - Garbage collection
  - Subclass **finalize** method
    - should invoke superclass **finalize** method

14

```java
1   // Fig. 9.7: Point.java
2   // Definition of class Point
3   public class Point extends Object {
4      protected int x, y; // coordinates of the Point
5
6      // no-argument constructor
7      public Point()
8      {
9         x = 0;
10        y = 0;
11        System.out.println( "Point constructor: " + this );
12     }
13
14     // constructor
15     public Point( int xCoordinate, int yCoordinate )
16     {
17        x = xCoordinate;
18        y = yCoordinate;
19        System.out.println( "Point constructor: " + this );
20     }
21
22     // finalizer
23     protected void finalize()
24     {
25        System.out.println( "Point finalizer: " + this );
26     }
27
28     // convert Point into a String representation
29     public String toString()
30     {
31        return "[" + x + ", " + y + "]";
32     }
33
34  }  // end class Point
```

Outline

`Point.java`

Lines 7-20
Superclass constructors

Lines 23-26
Superclass **`finalize`** method uses **`protected`** for subclass access, but not for other clients

15

```java
1   // Fig. 9.8: Circle.java
2   // Definition of class Circle
3   public class Circle extends Point {  // inherits from Point
4      protected double radius;
5
6      // no-argument constructor
7      public Circle()
8      {
9         // implicit call to superclass constructor here
10        radius = 0;
11        System.out.println( "Circle constructor: " + this );
12     }
13
14     // Constructor
15     public Circle( double circleRadius, int xCoordinate,
16        int yCoordinate )
17     {
18        // call superclass constructor
19        super( xCoordinate, yCoordinate );
20
21        radius = circleRadius;
22        System.out.println( "Circle constructor: " + this );
23     }
24
25     // finalizer
26     protected void finalize()
27     {
28        System.out.println( "Circle finalizer: " + this );
29        super.finalize();  // call superclass finalize method
30     }
31
```

Outline

`Circle.java`

Line 9
Implicit call to **`Point`** constructor

Line 19
Explicit call to **`Point`** constructor using **`super`**

Lines 26-30
Override **`Point`**'s method **`finalize`**, but call it using **`super`**

16

Outline

Circle.java

```java
32     // convert the Circle to a String
33     public String toString()
34     {
35        return "Center = " + super.toString() +
36               "; Radius = " + radius;
37     }
38
39  }  // end class Circle
```


Outline

Test.java

Lines 10-11
Instantiate **Circle** objects

Line 17
Invoke **Circle**'s method **finalize** by calling **System.gc**

```java
1   // Fig. 9.9: Test.java
2   // Demonstrate when superclass and subclass
3   // constructors and finalizers are called.
4   public class Test {
5
6      // test when constructors and finalizers are called
7      public static void main( String args[] )
8      {
9         Circle circle1, circle2;
10
11        circle1 = new Circle( 4.5, 72, 29 );
12        circle2 = new Circle( 10, 5, 5 );
13
14        circle1 = null;  // mark for garbage collection
15        circle2 = null;  // mark for garbage collection
16
17        System.gc();     // call the garbage collector
18     }
19
20  }  // end class Test
```

```
Point constructor: Center = [72, 29]; Radius = 0.0
Circle constructor: Center = [72, 29]; Radius = 4.5
Point constructor: Center = [5, 5]; Radius = 0.0
Circle constructor: Center = [5, 5]; Radius = 10.0
Circle finalizer: Center = [72, 29]; Radius = 4.5
Point finalizer: Center = [72, 29]; Radius = 4.5
Circle finalizer: Center = [5, 5]; Radius = 10.0
Point finalizer: Center = [5, 5]; Radius = 10.0
```

Implicit Subclass-Object-to-Superclass-Object Conversion

- Superclass reference and subclass reference
  - Implicit conversion
    - Subclass reference to superclass reference
      - Subclass object "is a" superclass object
  - Four ways to mix and match references
    - Refer to superclass object with superclass reference
    - Refer to subclass object with subclass reference
    - Refer to subclass object with superclass reference
      - **Can refer only to superclass members**
      - ➔ **Superclass = subclass**
    - Refer to superclass object with subclass reference
      - **Syntax error**
      - ➔ **Subclass = superclass is error!!!! ➔ should convert the type**

19

Software Engineering with Inheritance

- Inheritance
  - Create class (subclass) from existing one (superclass)
    - Subclass creation does not affect superclass
  - New class inherits attributes and behaviors of superclass
  - Add attributes and behaviors or override superclass behavior
  - Software reuse

Composition vs. Inheritance

- Inheritance
  - "Is a" relationship
  - `Teacher` *is an* `Employee`
- Composition
  - "Has a" relationship
  - `Employee` *has a* `TelephoneNumber`

20

Case Study: Point, Cylinder, Circle

- Consider point, circle, cylinder hierarchy
  - **`Point`** is superclass
  - **`Circle`** is **`Point`** subclass
  - **`Cylinder`** is **`Circle`** subclass


```
1   // Fig. 9.10: Point.java
2   // Definition of class Point
3   package com.deitel.jhtp4.ch09;
4
5   public class Point {
6      protected int x, y; // coordinates of Point
7
8      // No-argument constructor
9      public Point()
10     {
11        // implicit call to superclass constructor occurs here
12        setPoint( 0, 0 );
13     }
14
15     // constructor
16     public Point( int xCoordinate, int yCoordinate )
17     {
18        // implicit call to superclass constructor occurs here
19        setPoint( xCoordinate, yCoordinate );
20     }
21
22     // set x and y coordinates of Point
23     public void setPoint( int xCoordinate, int yCoordinate )
24     {
25        x = xCoordinate;
26        y = yCoordinate;
27     }
28
29     // get x coordinate
30     public int getX()
31     {
32        return x;
33     }
34
```

Outline

`Point.java`

Line 6
`protected` members prevent clients from direct access (unless clients are **`Point`** subclasses or are in same package)

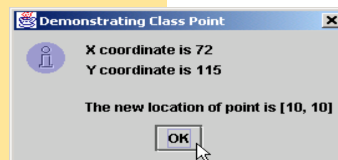
Lines 9-20
Constructor and overloaded constructor

```
35      // get y coordinate
36      public int getY()
37      {
38         return y;
39      }
40
41      // convert into a String representation
42      public String toString()
43      {
44         return "[" + x + ", " + y + "]";
45      }
46
47   }  // end class Point
```

Outline

`Point.java`


```
1    // Fig. 9.11: Test.java
2    // Applet to test class Point
3
4    // Java extension packages
5    import javax.swing.JOptionPane;
6
7    // Deitel packages
8    import com.deitel.jhtp4.ch09.Point;
9
10   public class Test {
11
12      // test class Point
13      public static void main( String args[] )
14      {
15         Point point = new Point( 72, 115 );
16
17         // get coordinates
18         String output = "X coordinate is " + point.getX() +
19                  "\nY coordinate is " + point.getY();
20
21         // set coordinates
22         point.setPoint( 10, 10 );
23
24         // use implicit call to point.toString()
25         output += "\n\nThe new location of point is " + point;
26
27         JOptionPane.showMessageDialog( null, output,
28            "Demonstrating Class Point",
29            JOptionPane.INFORMATION_MESSAGE );
30
31         System.exit( 0 );
32      }
33
34   }  // end class Test
```



Outline

`Test.java`

Line 15
Instantiate **`Point`** object

Lines 18-19
Methods **`getX`** and **`getY`** read **`Point`**'s **`protected`** variables

```
1    // Fig. 9.12: Circle.java
2    // Definition of class Circle
3    package com.deitel.jhtp4.ch09;
4
5    public class Circle extends Point {  // inherits from Point
6       protected double radius;
7
8       // no-argument constructor
9       public Circle()
10      {
11         // implicit call to superclass constructor occurs here
12         setRadius( 0 );
13      }
14
15      // constructor
16      public Circle( double circleRadius, int xCoordinate,
17         int yCoordinate )
18      {
19         // call superclass constructor to set coordinates
20         super( xCoordinate, yCoordinate );
21
22         // set radius
23         setRadius( circleRadius );
24      }
25
26      // set radius of Circle
27      public void setRadius( double circleRadius )
28      {
29         radius = ( circleRadius >= 0.0 ? circleRadius : 0.0 );
30      }
31
```

Outline

Circle.java

Line 5
Circle is a **Point** subclass

Line 5
Circle inherits **Point**'s **protected** variables and **public** methods (except for constuctor)

Line 11
Implicit call to **Point** constructor

Line 20
explicit call to **Point** constructor using **super**


```
32      // get radius of Circle
33      public double getRadius()
34      {
35         return radius;
36      }
37
38      // calculate area of Circle
39      public double area()
40      {
41         return Math.PI * radius * radius;
42      }
43
44      // convert the Circle to a String
45      public String toString()
46      {
47         return "Center = " + "[" + x + ", " + y + "]" +
48                "; Radius = " + radius;
49      }
50
51   }  // end class Circle
```

Outline

Circle.java

```
1    // Fig. 9.13: Test.java
2    // Applet to test class Circle
3
4    // Java core packages
5    import java.text.DecimalFormat;
6
7    // Java extension packages
8    import javax.swing.JOptionPane;
9
10   // Deitel packages
11   import com.deitel.jhtp4.ch09.Circle;
12
13   public class Test {
14
15      // test class Circle
16      public static void main( String args[] )
17      {
18         // create a Circle
19         Circle circle = new Circle( 2.5, 37, 43 );
20         DecimalFormat precision2 = new DecimalFormat( "0.00" );
21
22         // get coordinates and radius
23         String output = "X coordinate is " + circle.getX() +
24                 "\nY coordinate is " + circle.getY() +
25                 "\nRadius is " + circle.getRadius();
26
27         // set coordinates and radius
28         circle.setRadius( 4.25 );
29         circle.setPoint( 2, 2 );
30
31         // get String representation of Circle and calculate area
32         output +=
33            "\n\nThe new location and radius of c are\n" + circle +
34            "\nArea is " + precision2.format( circle.area() );
35
```

Outline

`Test.java`

Line 19
Instantiate **`Circle`** object

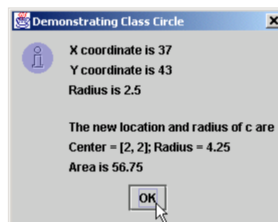
Lines 25 and 28
Calls to methods **`getRadius`** and **`setRadius`** read and manipulate **`Circle`**'s **`protected`** variables

Lines 23-24 and 29
Calls to methods **`getX`**, **`getY`** and **`setPoint`** read and manipulate **`Circle`**'s inherited **`protected`** variables


```
36         JOptionPane.showMessageDialog( null, output,
37            "Demonstrating Class Circle",
38            JOptionPane.INFORMATION_MESSAGE );
39
40         System.exit( 0 );
41      }
42
43   }  // end class Test
```

Outline

`Test.java`



```
1   // Fig. 9.14: Cylinder.java
2   // Definition of class Cylinder
3   package com.deitel.jhtp4.ch09;
4
5   public class Cylinder extends Circle {
6      protected double height;  // height of Cylinder
7
8      // no-argument constructor
9      public Cylinder()
10     {
11        // implicit call to superclass constructor here
12        setHeight( 0 );
13     }
14
15     // constructor
16     public Cylinder( double cylinderHeight, double cylinderRadius,
17        int xCoordinate, int yCoordinate )
18     {
19        // call superclass constructor to set coordinates/radius
20        super( cylinderRadius, xCoordinate, yCoordinate );
21
22        // set cylinder height
23        setHeight( cylinderHeight );
24     }
25
26     // set height of Cylinder
27     public void setHeight( double cylinderHeight )
28     {
29        height = ( cylinderHeight >= 0 ? cylinderHeight : 0 );
30     }
31
```

Outline

`Cylinder.java`

Line 5
`Cylinder` is a **`Circle`** subclass

Line 5
`Cylinder` inherits **`Point`**'s and **`Circle`**'s **`protected`** variables and **`public`** methods (except for constuctors)

Line 11
Implicit call to **`Circle`** constructor

Line 20
Explicit call to **`Circle`** constructor using **`super`**

29

```
32     // get height of Cylinder
33     public double getHeight()
34     {
35        return height;
36     }
37
38     // calculate area of Cylinder (i.e., surface area)
39     public double area()
40     {
41        return 2 * super.area() +
42               2 * Math.PI * radius * height;
43     }
44
45     // calculate volume of Cylinder
46     public double volume()
47     {
48        return super.area() * height;
49     }
50
51     // convert the Cylinder to a String
52     public String toString()
53     {
54        return super.toString() + "; Height = " + height;
55     }
56
57  }  // end class Cylinder
```

Outline

`Cylinder.java`

Lines 39-43
Override method area of class **`Circle`**

30

Outline

`Test.java`

Line 19
Instantiate **`Cylinder`** object

Lines 23-31
Method calls read and manipulate **`Cylinder`**'s **`protected`** variables and inherited **`protected`** variables

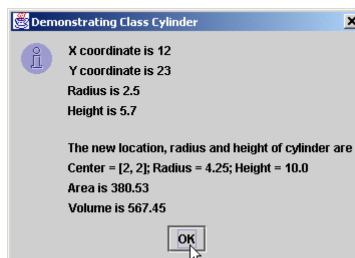
```
1   // Fig. 9.15: Test.java
2   // Application to test class Cylinder
3
4   // Java core packages
5   import java.text.DecimalFormat;
6
7   // Java extension packages
8   import javax.swing.JOptionPane;
9
10  // Deitel packages
11  import com.deitel.jhtp4.ch09.Cylinder;
12
13  public class Test {
14
15     // test class Cylinder
16     public static void main( String args[] )
17     {
18        // create Cylinder
19        Cylinder cylinder = new Cylinder( 5.7, 2.5, 12, 23 );
20        DecimalFormat precision2 = new DecimalFormat( "0.00" );
21
22        // get coordinates, radius and height
23        String output = "X coordinate is " + cylinder.getX() +
24           "\nY coordinate is " + cylinder.getY() +
25           "\nRadius is " + cylinder.getRadius() +
26           "\nHeight is " + cylinder.getHeight();
27
28        // set coordinates, radius and height
29        cylinder.setHeight( 10 );
30        cylinder.setRadius( 4.25 );
31        cylinder.setPoint( 2, 2 );
32
```

31

Outline

`Test.java`

```
33        // get String representation of Cylinder and calculate
34        // area and volume
35        output += "\n\nThe new location, radius " +
36           "and height of cylinder are\n" + cylinder +
37           "\nArea is " + precision2.format( cylinder.area() ) +
38           "\nVolume is " + precision2.format( cylinder.volume() );
39
40        JOptionPane.showMessageDialog( null, output,
41           "Demonstrating Class Cylinder",
42           JOptionPane.INFORMATION_MESSAGE );
43
44        System.exit( 0 );
45     }
46
47  }  // end class Test
```



32

Introduction to Polymorphism

- Polymorphism
  - Helps build extensible systems
  - Programs generically process objects as superclass objects
    - Can add classes to systems easily
      - Classes must be part of generically processed hierarchy

Type Fields and Switch Statements

- **`switch`**-based system
  - Determine appropriate action for object of many different types
    - Based on object's type to take the appropriate action
  - Error prone
    - Programmer can forget to make appropriate type test
    - Programmer may forget to Adding and deleting new case in existing **`switch`** statements

33

Dynamic Method Binding

- Dynamic method binding
  - Implements polymorphic processing of objects
  - Use superclass reference to refer to subclass object
  - Program chooses "correct" method in subclass
- For example,
  - Superclass **`Shape`**
  - Subclasses **`Circle`**, **`Rectangle`** and **`Square`**
  - Each class draws itself according to type of class
    - **`Shape`** has method **`draw`**
    - Each subclass overrides method **`draw`**
    - Call method **`draw`** of superclass **`Shape`**
      - Program determines dynamically which subclass **`draw`** method to invoke

34

final Methods and Classes

- `final` method
  - Cannot be overridden in subclass
- `final` class
  - Cannot be superclass (cannot be `extend`ed)
    - Class cannot inherit `final` classes
  - Final method's definition can never change
  - The compiler can optimize the program by removing calls to final method and replace them with the expanded code of their definitions at each method call location ➔ in-lining the code(like assembly's macro instruction)

35

Abstract Superclasses and Concrete Classes

- Abstract classes
  - Objects cannot be instantiated➔ programmer never intends to use to instantiate any objects
  - Too generic to define real objects
    - **TwoDimensionalShape in pp. 451**
  - Provides superclass from which other classes may inherit
    - Normally referred to as ***abstract super-classes***
  - ***Draw the shape ➔ what shape?***
- Concrete classes
  - Classes from which objects are instantiated
  - Provide specifics for instantiating objects
    - **Square**, **Circle** and **Triangle in pp. 451 ➔ derived concrete class**
  - ***Draw the shape ➔ draw the* Square**, **Circle** and **Triangle** ***shape?***

36

Polymorphism Examples

- Video game
  - Superclass **`GamePiece`**
    - Contains method **`drawYourself`**
  - Subclasses **`Martian`**, **`Venutian`**, **`LaserBeam`**, etc.
    - Override method **`drawYourself`**
      - **`Martian`** draws itself with antennae
      - **`LaserBeam`** draws itself as bright red beam
      - This is polymorphism
  - Easily extensible
    - Suppose we add class **`Mercurian`**
      - Class **`Mercurian`** inherits superclass **`GamePiece`**
      - Overrides method **`drawYourself`**

37

Case Study: A Payroll System Using Polymorphism

- Abstract methods and polymorphism
  - Abstract superclass **Employee**
    - Method **earnings** applies to all employees
    - Person's earnings dependent on type of **Employee**
    - **Earnings is declared abstract in super-class Employee and appropriate implementations of earnings are provided for each of the super-class**
  - Concrete **Abstract superclass Employee** subclasses are
    - **Boss**
    - **CommissionWorker**
    - **PieceWorker**
    - **HourlyWorker**

38

```
1    // Fig. 9.16: Employee.java
2    // Abstract base class Employee.
3
4    public abstract class Employee {
5       private String firstName;
6       private String lastName;
7
8       // constructor
9       public Employee( String first, String last )
10      {
11         firstName = first;
12         lastName = last;
13      }
14
15      // get first name
16      public String getFirstName()
17      {
18         return firstName;
19      }
20
21      // get last name
22      public String getLastName()
23      {
24         return lastName;
25      }
26
27      public String toString()
28      {
29         return firstName + ' ' + lastName;
30      }
32      // Abstract method that must be implemented for each
33      // derived class of Employee from which objects are instantiated.
35      public abstract double earnings();
36   }  // end class Employee
```

Outline

Employee.java

Line 4
abstract class cannot be instantiated

Lines 5-6 and 16-30
abstract class can have instance data and non**abstract** methods for subclasses

Lines 9-13
abstract class can have constructors for subclasses to initialize inherited data


```
1    // Fig. 9.17: Boss.java
2    // Boss class derived from Employee.
3
4    public final class Boss extends Employee {
5       private double weeklySalary;
6
7       // constructor for class Boss
8       public Boss( String first, String last, double salary )
9       {
10         super( first, last );  // call superclass constructor
11         setWeeklySalary( salary );
12      }
13
14      // set Boss's salary
15      public void setWeeklySalary( double salary )
16      {
17         weeklySalary = ( salary > 0 ? salary : 0 );
18      }
19
20      // get Boss's pay
21      public double earnings()
22      {
23         return weeklySalary;
24      }
25
26      // get String representation of Boss's name
27      public String toString()
28      {
29         return "Boss: " + super.toString();
30      }
31
32   }  // end class Boss
```

Outline

Boss.java

Line 4
Boss is an **Employee** subclass

Line 4
Boss inherits **Employee**'s **public** methods (except for constuctor)

Line 10
Explicit call to **Employee** constructor using **super**

Lines 21-24
Required to implement **Employee**'s method **earnings** (polymorphism)

```java
1    // Fig. 9.18: CommissionWorker.java
2    // CommissionWorker class derived from Employee
3
4    public final class CommissionWorker extends Employee {
5       private double salary;      // base salary per week
6       private double commission;  // amount per item sold
7       private int quantity;       // total items sold for week
8
9       // constructor for class CommissionWorker
10      public CommissionWorker( String first, String last,
11         double salary, double commission, int quantity )
12      {
13         super( first, last );  // call superclass constructor
14         setSalary( salary );
15         setCommission( commission );
16         setQuantity( quantity );
17      }
18
19      // set CommissionWorker's weekly base salary
20      public void setSalary( double weeklySalary )
21      {
22         salary = ( weeklySalary > 0 ? weeklySalary : 0 );
23      }
24
25      // set CommissionWorker's commission
26      public void setCommission( double itemCommission )
27      {
28         commission = ( itemCommission > 0 ? itemCommission : 0 );
29      }
30
```

Outline

CommissionWorker.java

Line 4
CommissionWorker is an **Employee** subclass

Line 13
Explicit call to **Employee** constructor using **super**

41

```java
31      // set CommissionWorker's quantity sold
32      public void setQuantity( int totalSold )
33      {
34         quantity = ( totalSold > 0 ? totalSold : 0 );
35      }
36
37      // determine CommissionWorker's earnings
38      public double earnings()
39      {
40         return salary + commission * quantity;
41      }
42
43      // get String representation of CommissionWorker's name
44      public String toString()
45      {
46         return "Commission worker: " + super.toString();
47      }
48
49   }  // end class CommissionWorker
```

Outline

CommissionWorker.java

Lines 38-41
Required to implement **Employee**'s method **earnings**; this implementation differs from that in **Boss**

42

```
1    // Fig. 9.19: PieceWorker.java
2    // PieceWorker class derived from Employee
3
4    public final class PieceWorker extends Employee {
5       private double wagePerPiece; // wage per piece output
6       private int quantity;        // output for week
7
8       // constructor for class PieceWorker
9       public PieceWorker( String first, String last,
10         double wage, int numberOfItems )
11      {
12         super( first, last );  // call superclass constructor
13         setWage( wage );
14         setQuantity( numberOfItems);
15    }
16
17      // set PieceWorker's wage
18      public void setWage( double wage )
19      {
20         wagePerPiece = ( wage > 0 ? wage : 0 );
21      }
22
23      // set number of items output
24      public void setQuantity( int numberOfItems )
25      {
26         quantity = ( numberOfItems > 0 ? numberOfItems : 0 );
27      }
28
29      // determine PieceWorker's earnings
30      public double earnings()
31      {
32         return quantity * wagePerPiece;
33      }
34
```

Outline

PieceWorker.java

Line 4
PieceWorker is an **Employee** subclass

Line 12
Explicit call to **Employee** constructor using **super**

Lines 30-33
Implementation of **Employee**'s method **earnings**; differs from that of **Boss** and **CommissionWorker**

43

```
35      public String toString()
36      {
37         return "Piece worker: " + super.toString();
38      }
39
40   }  // end class PieceWorker
```

Outline

PieceWorker.java

44

Outline

`HourlyWorker.java`

Line 4
`PieceWorker` is an **`Employee`** subclass

Line 12
Explicit call to **`Employee`** constructor using **`super`**

Line 31
Implementation of **`Employee`**'s method **`earnings`**; differs from that of other **`Employee`** subclasses

```
1    // Fig. 9.20: HourlyWorker.java
2    // Definition of class HourlyWorker
3
4    public final class HourlyWorker extends Employee {
5       private double wage;    // wage per hour
6       private double hours;   // hours worked for week
7
8       // constructor for class HourlyWorker
9       public HourlyWorker( String first, String last,
10         double wagePerHour, double hoursWorked )
11      {
12         super( first, last );   // call superclass constructor
13         setWage( wagePerHour );
14         setHours( hoursWorked );
15      }
16
17      // Set the wage
18      public void setWage( double wagePerHour )
19      {
20         wage = ( wagePerHour > 0 ? wagePerHour : 0 );
21      }
22
23      // Set the hours worked
24      public void setHours( double hoursWorked )
25      {
26         hours = ( hoursWorked >= 0 && hoursWorked < 168 ?
27            hoursWorked : 0 );
28      }
29
30      // Get the HourlyWorker's pay
31      public double earnings() { return wage * hours; }
32
```

45

Outline

`HourlyWorker.java`

```
33      public String toString()
34      {
35         return "Hourly worker: " + super.toString();
36      }
37
38   }  // end class HourlyWorker
```

46

```
1     // Fig. 9.21: Test.java
2     // Driver for Employee hierarchy
3
4     // Java core packages
5     import java.text.DecimalFormat;
6
7     // Java extension packages
8     import javax.swing.JOptionPane;
9
10    public class Test {
11
12       // test Employee hierarchy
13       public static void main( String args[] )
14       {
15          Employee employee;  // superclass reference
16          String output = "";
17
18          Boss boss = new Boss( "John", "Smith", 800.0 );
19
20           CommissionWorker commisionWorker =
21             new CommissionWorker(
22             "Sue", "Jones", 400.0, 3.0, 150 );
23
24         PieceWorker pieceWorker =
25           new PieceWorker( "Bob", "Lewis", 2.5, 200 );
26
27         HourlyWorker hourlyWorker =
28            new HourlyWorker( "Karen", "Price", 13.75, 40 );
29
30        DecimalFormat precision2 = new DecimalFormat( "0.00" );
31
```

Outline

Test.java

Line 15
Test cannot instantiate **Employee** but can reference one

Lines 18-28
Instantiate one instance each of **Employee** subclasses


```
32        // Employee reference to a Boss
33        employee = boss;
34
35        output += employee.toString() + " earned $" +
36           precision2.format( employee.earnings() ) + "\n" +
37           boss.toString() + " earned $" +
38           precision2.format( boss.earnings() ) + "\n";
39
40        // Employee reference to a CommissionWorker
41        employee = commissionWorker;
42
43        output += employee.toString() + " earned $" +
44           precision2.format( employee.earnings() ) + "\n" +
45           commissionWorker.toString() + " earned $" +
46           precision2.format(
47              commissionWorker.earnings() ) + "\n";
48
49        // Employee reference to a PieceWorker
50        employee = pieceWorker;
51
52        output += employee.toString() + " earned $" +
53           precision2.format( employee.earnings() ) + "\n" +
54           pieceWorker.toString() + " earned $" +
55           precision2.format( pieceWorker.earnings() ) + "\n";
56
```

Outline

Test.java

Line 33
Use **Employee** to reference **Boss**

Line 36
Method **employee.earnings** dynamically binds to method **boss.earnings**

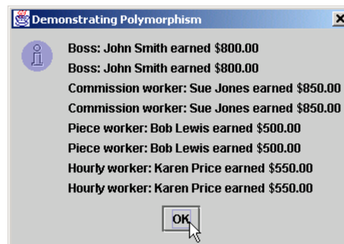
Lines 41-55
Do same for **CommissionWorker** and **PieceWorker**

```
57       // Employee reference to an HourlyWorker
58       employee = hourlyWorker;
59
60       output += employee.toString() + " earned $" +
61       precision2.format( employee.earnings() ) + "\n" +
62            hourlyWorker.toString() + " earned $" +
63            precision2.format( hourlyWorker.earnings() ) + "\n";
64
65         JOptionPane.showMessageDialog( null, output,
66            "Demonstrating Polymorphism",
67            JOptionPane.INFORMATION_MESSAGE );
68
69         System.exit( 0 );
70      }
71
72   }  // end class Test
```

Outline

Test.java

Lines 58-63
Repeat for **HourlyWorker**



49

New Classes and Dynamic Binding

- Dynamic binding (late binding)
  - Object's type need not be know at compile time
  - At run time, call is matched with method of called object
- **Point**, **Circle**, **Cylinder** hierarchy
  - Modify by including **abstract** superclass **Shape**
    - Demonstrates polymorphism
    - Contains **abstract** method **getName**
      - Each subclass must implement method **getName**
    - Contains (non**abstract**) methods **area** and **volume**
      - **Return 0 by default**
      - Each subclass **overrides** these methods

50

Outline

Case Study: Inheriting Interface and Implementation

```
1    // Fig. 9.22: Shape.java
2    // Definition of abstract base class Shape
3
4    public abstract class Shape extends Object {
5
6       // return shape's area
7       public double area()
8       {
9          return 0.0;
10      }
11
12      // return shape's volume
13      public double volume()
14      {
15         return 0.0;
16      }
17
18      // abstract method must be defined by concrete subclasses
19      // to return appropriate shape name
20      public abstract String getName();
21
22   }  // end class Shape
```

Shape.java

Line 4
Shape cannot be instantiated

Lines 7-16
abstract class can have non**abstract** methods for subclasses

Line 20
Concrete subclasses must implement method **getName**


```
1    // Fig. 9.23: Point.java
2    // Definition of class Point
3
4    public class Point extends Shape {
5       protected int x, y; // coordinates of the Point
6
7       // no-argument constructor
8       public Point()
9       {
10         setPoint( 0, 0 );
11      }
12
13      // constructor
14      public Point( int xCoordinate, int yCoordinate )
15      {
16         setPoint( xCoordinate, yCoordinate );
17      }
18
19      // set x and y coordinates of Point
20      public void setPoint( int xCoordinate, int yCoordinate )
21      {
22         x = xCoordinate;
23         y = yCoordinate;
24      }
25
26      // get x coordinate
27      public int getX()
28      {
29         return x;
30      }
31
```

Outline

Point.java

Line 4
Point inherits **Shape**'s **public** methods

Line 5
protected members prevent clients from direct access (unless clients are **Point** subclasses or are in same package)

```
32      // get y coordinate
33      public int getY()
34      {
35         return y;
36      }
37
38      // convert point into String representation
39      public String toString()
40      {
41         return "[" + x + ", " + y + "]";
42      }
43
44      // return shape name
45      public String getName()
46      {
47         return "Point";
48      }
49
50   }  // end class Point
```

Outline

Point.java

Lines 45-48
Implementation of **Shape**'s method **getName**

*** **Note** ***


```
1    // Fig. 9.24: Circle.java
2    // Definition of class Circle
3
4    public class Circle extends Point {  // inherits from Point
5       protected double radius;
6
7       // no-argument constructor
8       public Circle()
9       {
10         // implicit call to superclass constructor here
11         setRadius( 0 );
12      }
13
14      // constructor
15      public Circle( double circleRadius, int xCoordinate,
16         int yCoordinate )
17      {
18         // call superclass constructor
19         super( xCoordinate, yCoordinate );
20
21         setRadius( circleRadius );
22      }
23
24      // set radius of Circle
25      public void setRadius( double circleRadius )
26      {
27         radius = ( circleRadius >= 0 ? circleRadius : 0 );
28      }
29
30      // get radius of Circle
31      public double getRadius()
32      {
33         return radius;
34      }
35
```

Outline

Circle.java

Line 4
Circle inherits variables/methods from **Point** and **Shape**

Lines 5 and 24-34
Methods for reading/setting **protected** value

```
36      // calculate area of Circle
37      public double area()
38      {
39         return Math.PI * radius * radius;
40      }
41
42      // convert Circle to a String represention
43      public String toString()
44      {
45         return "Center = " + super.toString() +
46            "; Radius = " + radius;
47      }
48
49      // return shape name
50      public String getName()
51      {
52         return "Circle";
53      }
54
55   }  // end class Circle
```

Outline

Circle.java

Lines 37-40
Override method **area** but not method **volume** (circles do not have volume)

Lines 50-53
Implementation of **Shape**'s method **getName**


```
1    // Fig. 9.25: Cylinder.java
2    // Definition of class Cylinder.
3
4    public class Cylinder extends Circle {
5       protected double height;  // height of Cylinder
6
7       // no-argument constructor
8       public Cylinder()
9       {
10         // implicit call to superclass constructor here
11         setHeight( 0 );
12      }
13
14      // constructor
15      public Cylinder( double cylinderHeight,
16         double cylinderRadius, int xCoordinate,
17         int yCoordinate )
18      {
19         // call superclass constructor
20         super( cylinderRadius, xCoordinate, yCoordinate );
21
22         setHeight( cylinderHeight );
23      }
24
25      // set height of Cylinder
26      public void setHeight( double cylinderHeight )
27      {
28         height = ( cylinderHeight >= 0 ? cylinderHeight : 0 );
29      }
30
```

Outline

Cylinder.java

Line 4
Cylinder inherits variables and methods from **Point**, **Circle** and **Shape**

```java
31    // get height of Cylinder
32    public double getHeight()
33    {
34       return height;
35    }
36
37    // calculate area of Cylinder (i.e., surface area)
38    public double area()
39    {
40       return 2 * super.area() + 2 * Math.PI * radius * height;
41    }
42
43    // calculate volume of Cylinder
44    public double volume()
45    {
46       return super.area() * height;
47    }
48
49    // convert Cylinder to a String representation
50    public String toString()
51    {
52       return super.toString() + "; Height = " + height;
53    }
54
55    // return shape name
56    public String getName()
57    {
58       return "Cylinder";
59    }
60
61 }  // end class Cylinder
```

Outline

Cylinder.java

Lines 38-47
Override methods **area** and **volume**

Lines 56-59
Implementation of **Shape**'s method **getName**


```java
1  // Fig. 9.26: Test.java
2  // Class to test Shape, Point, Circle, Cylinder hierarchy
3
4  // Java core packages
5  import java.text.DecimalFormat;
6
7  // Java extension packages
8  import javax.swing.JOptionPane;
9
10 public class Test {
11
12    // test Shape hierarchy
13    public static void main( String args[] )
14    {
15       // create shapes
16       Point point = new Point( 7, 11 );
17       Circle circle = new Circle( 3.5, 22, 8 );
18       Cylinder cylinder = new Cylinder( 10, 3.3, 10, 10 );
19
20       // create Shape array
21       Shape arrayOfShapes[] = new Shape[ 3 ];
22
23       // aim arrayOfShapes[ 0 ] at subclass Point object
24       arrayOfShapes[ 0 ] = point;
25
26       // aim arrayOfShapes[ 1 ] at subclass Circle object
27       arrayOfShapes[ 1 ] = circle;
28
29       // aim arrayOfShapes[ 2 ] at subclass Cylinder object
30       arrayOfShapes[ 2 ] = cylinder;
31
```

Outline

Test.java

Lines 16-18
Instantiate one instance each of **Shape** subclasses

Lines 21-30
Create three **Shape**s to reference each subclass object

```
32        // get name and String representation of each shape
33        String output =
34           point.getName() + ": " + point.toString() + "\n" +
35           circle.getName() + ": " + circle.toString() + "\n" +
36           cylinder.getName() + ": " + cylinder.toString();
37
38        DecimalFormat precision2 = new DecimalFormat( "0.00" );
39
40        // loop through arrayOfShapes and get name,
41        // area and volume of each shape in arrayOfShapes
42        for ( int i = 0; i < arrayOfShapes.length; i++ ) {
43           output += "\n\n" + arrayOfShapes[ i ].getName() +
44              ": " + arrayOfShapes[ i ].toString() +
45              "\nArea = " +
46              precision2.format( arrayOfShapes[ i ].area() ) +
47              "\nVolume = " +
48              precision2.format( arrayOfShapes[ i ].volume() );
49        }
50
51        JOptionPane.showMessageDialog( null, output,
52           "Demonstrating Polymorphism",
53           JOptionPane.INFORMATION_MESSAGE );
54
55        System.exit( 0 );
56     }
57
58  }  // end class Test
```

Outline

`Test.java`

Line 43
Dynamically bind method **`getName`**

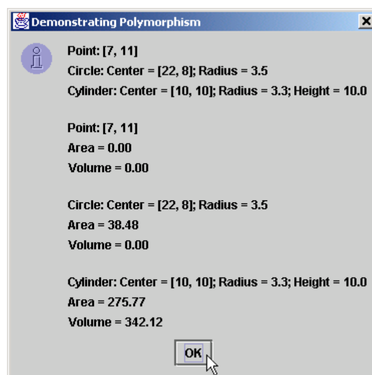
Line 46
Dynamically bind method **`area`** for **`Circle`** and **`Cylinder`** objects

Line 48
Dynamically bind method **`volume`** for **`Cylinder`** object

59

Outline

`Test.java`



60

Case Study: Creating and Using Interfaces

- Use **`interface Shape`**
  - Replace **`abstract`** class **`Shape`**
- Interface
  - Definition begins with ***`interface`*** keyword
  - Classes ***`implement`*** an interface (and its methods)
  - Contains ***`public abstract`*** methods
    - Classes (that **`implement`** the interface) must implement these methods


```
1   // Fig. 9.27: Shape.java
2   // Definition of interface Shape
3
4   public interface Shape {
5
6      // calculate area
7      public abstract double area();
8
9      // calculate volume
10     public abstract double volume();
11
12     // return shape name
13     public abstract String getName();
14  }
```

Outline

`Shape.java`

Lines 7-13
Classes that **`implement Shape`** must implement these methods

```
1     // Fig. 9.28: Point.java
2     // Definition of class Point
3
4     public class Point extends Object implements Shape {
5        protected int x, y; // coordinates of the Point
6
7        // no-argument constructor
8        public Point()
9        {
10          setPoint( 0, 0 );
11       }
12
13       // constructor
14       public Point( int xCoordinate, int yCoordinate )
15       {
16          setPoint( xCoordinate, yCoordinate );
17       }
18
19       // Set x and y coordinates of Point
20       public void setPoint( int xCoordinate, int yCoordinate )
21       {
22          x = xCoordinate;
23          y = yCoordinate;
24       }
25
26       // get x coordinate
27       public int getX()
28       {
29          return x;
30       }
31
```

Outline

`Point.java`

Line 4
`Point` implements interface **`Shape`**


```
32       // get y coordinate
33       public int getY()
34       {
35          return y;
36       }
37
38       // convert point into String representation
39       public String toString()
40       {
41          return "[" + x + ", " + y + "]";
42       }
43
44       // calculate area
45       public double area()
46       {
47          return 0.0;
48       }
49
50       // calculate volume
51       public double volume()
52       {
53          return 0.0;
54       }
55
56       // return shape name
57       public String getName()
58       {
59          return "Point";
60       }
61
62    }  // end class Point
```

Outline

`Point.java`

Lines 45-60
Implement methods specified by interface **`Shape`**

```java
1    // Fig. 9.29: Circle.java
2    // Definition of class Circle
3
4    public class Circle extends Point {  // inherits from Point
5       protected double radius;
6
7       // no-argument constructor
8       public Circle()
9       {
10         // implicit call to superclass constructor here
11         setRadius( 0 );
12      }
13
14      // constructor
15      public Circle( double circleRadius, int xCoordinate,
16         int yCoordinate )
17      {
18         // call superclass constructor
19         super( xCoordinate, yCoordinate );
20
21         setRadius( circleRadius );
22      }
23
24      // set radius of Circle
25      public void setRadius( double circleRadius )
26      {
27         radius = ( circleRadius >= 0 ? circleRadius : 0 );
28      }
29
30      // get radius of Circle
31      public double getRadius()
32      {
33         return radius;
34      }
35
```

Outline

`Circle.java`

Line 4
`Circle` inherits variables/methods from **`Point`**, including method implementations of **`Shape`**


```java
36      // calculate area of Circle
37      public double area()
38      {
39         return Math.PI * radius * radius;
40      }
41
42      // convert Circle to a String represention
43      public String toString()
44      {
45         return "Center = " + super.toString() +
46            "; Radius = " + radius;
47      }
48
49      // return shape name
50      public String getName()
51      {
52         return "Circle";
53      }
54
55   }  // end class Circle
```

Outline

`Circle.java`

Lines 43-47
Override method **`toString`**

Lines 37-40 and 50-53
Override methods **`area`** and **`getName`** but not method **`volume`**

Outline

`Cylinder.java`

Line 4
`Circle` inherits variables/methods from **`Point`** and **`Circle`** and method implementations of **`Shape`**

```
1    // Fig. 9.30: Cylinder.java
2    // Definition of class Cylinder.
3
4    public class Cylinder extends Circle {
5       protected double height;  // height of Cylinder
6
7       // no-argument constructor
8       public Cylinder()
9       {
10         // implicit call to superclass constructor here
11         setHeight( 0 );
12      }
13
14      // constructor
15      public Cylinder( double cylinderHeight,
16         double cylinderRadius, int xCoordinate,
17         int yCoordinate )
18      {
19         // call superclass constructor
20         super( cylinderRadius, xCoordinate, yCoordinate );
21
22         setHeight( cylinderHeight );
23      }
24
25      // set height of Cylinder
26      public void setHeight( double cylinderHeight )
27      {
28         height = ( cylinderHeight >= 0 ? cylinderHeight : 0 );
29      }
30
31      // get height of Cylinder
32      public double getHeight()
33      {
34         return height;
35      }
```

67

Outline

`Cylinder.java`

Lines 50-53
Override method **`toString`**

Line 38-59
Override methods **`area`**, **`volume`** and **`getName`**

```
36
37      // calculate area of Cylinder (i.e., surface area)
38      public double area()
39      {
40         return 2 * super.area() + 2 * Math.PI * radius * height;
41      }
42
43      // calculate volume of Cylinder
44      public double volume()
45      {
46         return super.area() * height;
47      }
48
49      // convert Cylinder to a String representation
50      public String toString()
51      {
52         return super.toString() + "; Height = " + height;
53      }
54
55      // return shape name
56      public String getName()
57      {
58         return "Cylinder";
59      }
60
61   }  // end class Cylinder
```

68

Outline

Test.java

Fig. 9.31 is identical to Fig. 9.26

```
1     // Fig. 9.31: Test.java
2     // Test Point, Circle, Cylinder hierarchy with interface Shape.
3
4     // Java core packages
5     import java.text.DecimalFormat;
6
7     // Java extension packages
8     import javax.swing.JOptionPane;
9
10    public class Test {
11
12       // test Shape hierarchy
13       public static void main( String args[] )
14       {
15          // create shapes
16          Point point = new Point( 7, 11 );
17          Circle circle = new Circle( 3.5, 22, 8 );
18          Cylinder cylinder = new Cylinder( 10, 3.3, 10, 10 );
19
20          // create Shape array
21          Shape arrayOfShapes[] = new Shape[ 3 ];
22
23          // aim arrayOfShapes[ 0 ] at subclass Point object
24          arrayOfShapes[ 0 ] = point;
25
26          // aim arrayOfShapes[ 1 ] at subclass Circle object
27          arrayOfShapes[ 1 ] = circle;
28
29          // aim arrayOfShapes[ 2 ] at subclass Cylinder object
30          arrayOfShapes[ 2 ] = cylinder;
31
```

69

Outline

Test.java

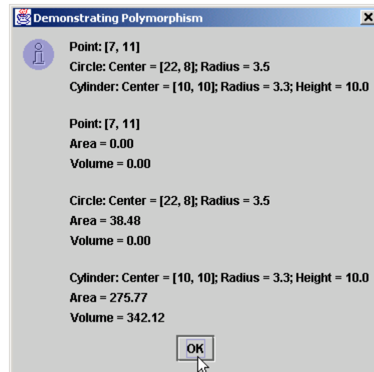
```
32          // get name and String representation of each shape
33          String output =
34             point.getName() + ": " + point.toString() + "\n" +
35             circle.getName() + ": " + circle.toString() + "\n" +
36             cylinder.getName() + ": " + cylinder.toString();
37
38          DecimalFormat precision2 = new DecimalFormat( "0.00" );
39
40          // loop through arrayOfShapes and get name,
41          // area and volume of each shape in arrayOfShapes
42          for ( int i = 0; i < arrayOfShapes.length; i++ ) {
43             output += "\n\n" + arrayOfShapes[ i ].getName() +
44                ": " + arrayOfShapes[ i ].toString() +
45                "\nArea = " +
46                precision2.format( arrayOfShapes[ i ].area() ) +
47                "\nVolume = " +
48                precision2.format( arrayOfShapes[ i ].volume() );
49          }
50
51          JOptionPane.showMessageDialog( null, output,
52             "Demonstrating Polymorphism",
53             JOptionPane.INFORMATION_MESSAGE );
54
55          System.exit( 0 );
56       }
57
58    }  // end class Test
```

70

Outline

Test.java

Output is identical to that of Fig. 9.26



71

Inner Class Definitions

- Inner classes
  - Class is defined inside another class body
  - Inner class object is allowed to access directly all the instance variable and methods of the outer class object.
  - Frequently used with GUI event handling
    - Declare **`ActionListener`** inner class
    - GUI components can register **`ActionListener`**s for events
      - Button events, key events, etc.

72

Inner Class Definitions (cont.)

- Anonymous inner class
  - Inner class without name
  - Created when class is defined in program

73

Notes on Inner Class Definitions

- Notes for inner-class definition and use
  - Compiling class that contains inner class
    - Results in separate **.class** file (outercslassname$innerclassname.class)
  - Inner classes with class names
    - **public**, **protected**, **private** or package access
  - Outer class is responsible for creating inner class objects
  - To create an object of another class's inner class, first create an object of the outer class and assign to a reference

  Ex. **OuterClassName.InnerClassName** innerRef=**ref.new InnerClassName();**
  - Inner classes can be declared **static→cannot access to the outer class's non-static members**

74

Type-Wrapper Classes for Primitive Types

- Type-wrapper class
  - Each primitive type has one
    - **Character**, **Byte**, **Integer**, **Boolean**, etc.
  - Enable to manipulate primitive types as **Object**
    - Value of primitive data types can be processed polymorphically
  - Declared as **final(Byte, Short, …)**→ inherit from **Number**
  - Many methods are declared final **static➔cannot be overriden**
- (Optional Case Study) Thinking About Objects: Incorporating Inheritance into the Elevator Simulation
- Our design can benefit from inheritance
  - Examine sets of classes
  - Look for commonality between/among sets ➔ Fig. 9.37
    - Extract commonality into superclass(UML's generalization)→ Location
      - Subclasses inherits this commonality(UML's specialization)→Elevator, Floor

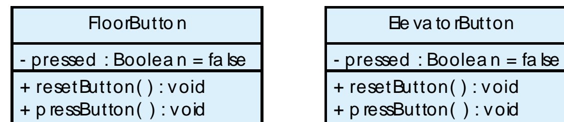
75

Thinking About Objects (cont.)

- **`ElevatorButton`** and **`FloorButton`**
  - Treated as separate classes
  - Both have *attribute* **`pressed`**
  - Both have *behaviors* **`pressButton`** and **`resetButton`**
  - Move attribute and behaviors into superclass **`Button`**?
    - We must examine whether these **objects have distinct behavior**
      - If **same behavior**
        - They are objects of class **`Button`**
      - If different behavior
        - They are objects of distinct **`Button`** subclasses

76

Fig. 9.35 Attributes and operations of classes **FloorButton** and **ElevatorButton.**



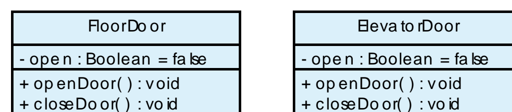
9.23 Thinking About Objects (cont.)

- **ElevatorButton** and **FloorButton**
  - **FloorButton** requests **Elevator** to **Floor** of request
    - **Elevator** will sometimes respond
  - **ElevatorButton** signals **Elevator** to move
    - **Elevator** will always respond
  - Neither button *decides* for the **Elevator** to move
    - **Elevator** decides itself
  - Both buttons signal **Elevator** to move
    - Therefore, **both buttons exhibit identical behavior**
      - They are objects of class **Button**
      - Combine (not inherit) **ElevatorButton** and **FloorButton** into class **Button**

9.23 Thinking About Objects (cont.)

- **`ElevatorDoor`** and **`FloorDoor`**
  - Treated as separate classes
  - Both have *attribute* **`open`**
  - Both have *behaviors* **`openDoor`** and **`closeDoor`**
  - Both door "inform" a **`Person`** that a door has opened
    - **Both doors exhibit identical behavior**
      - They are objects of class **`Door`**
      - Combine (not inherit) **`ElevatorDoor`** and **`FloorDoor`** into class **`Door`**

Fig. 9.36 Attributes and operations of classes FloorDoor and ElevatorDoor

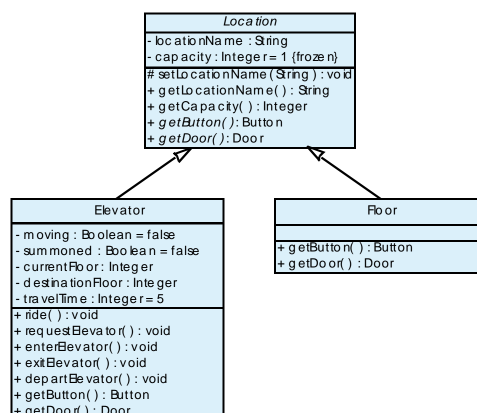


Thinking About Objects (cont.)

- Representing location of **`Person`**
  - On what **`Floor`** is **`Person`** when riding **`Elevator`**?
  - Both **`Floor`** and **`Elevator`** are types of locations
    - Share **`int`** attribute **`capacity`**
    - Inherit from **`abstract`** superclass **`Location`**
      - Contains **`String locationName`** representing location
        - **`"firstFloor"`**
        - **`"secondFloor"`**
        - **`"elevator"`**
  - **`Person`** now contains <u>**`Location`** reference</u>
    - References **`Elevator`** when person is <u>in elevator</u>
    - References **`Floor`** when person is <u>on floor</u>

81

Fig. 9.37 Class diagram modeling generalization of superclass **`Location`** and subclasses **`Elevator`** and **`Floor`**.



Outline

Italics indicate abstract class or method

Pound sign (**#**) indicates **`protected`** member

`{frozen}` indicates constant (final in Java)

Concrete classes implement abstract methods

82

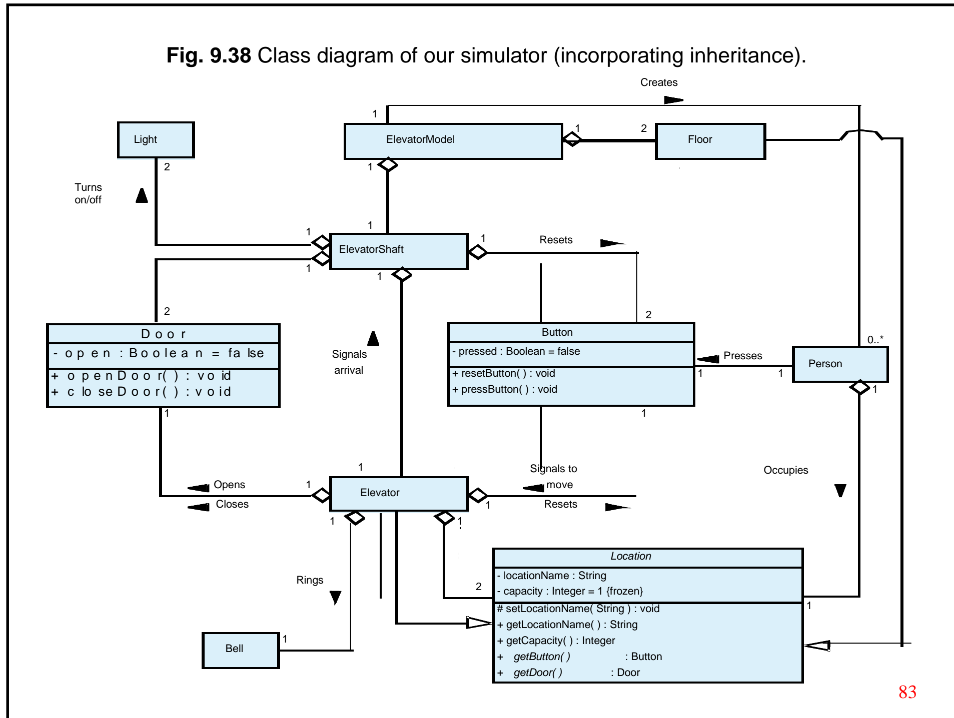


Fig. 9.38 Class diagram of our simulator (incorporating inheritance).

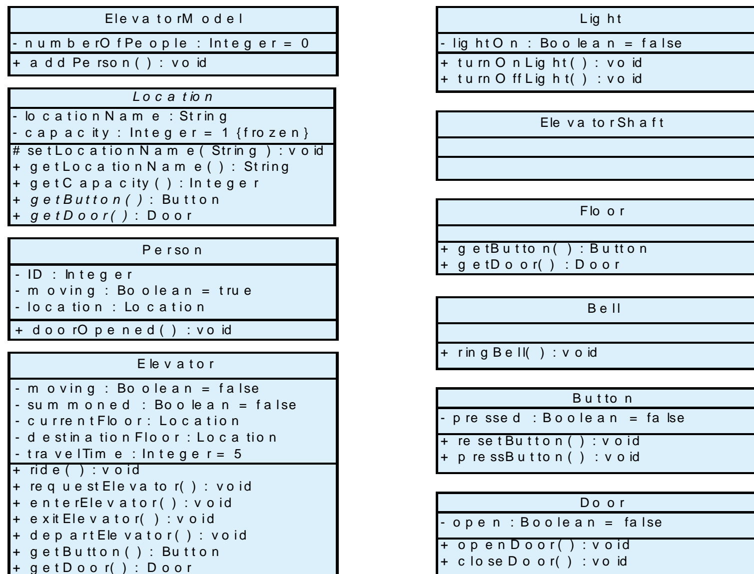


Fig. 9.39 Class diagram with attributes and operations (incorporating inheritance).

Thinking About Objects (cont.)

- Continue implementation
  - Transform design (i.e., class diagram) to code
  - Generate "skeleton code" with our design
    - Use class `Elevator` as example
    - Two steps (incorporating inheritance)

85

Thinking About Objects (cont.) Step 1

```
public class Elevator extends Location {

   // class constructor
   public Elevator() {}
}
```

86

```java
1   // Elevator.java
2   // Generated using class diagrams 9.38 and 9.39
3   public class Elevator extends Location {
4
5      // class attributes
6      private boolean moving;
7      private boolean summoned;
8      private Location currentFloor;
9      private Location destinationFloor;
10     private int travelTime = 5;
11     private Button elevatorButton;
12     private Door elevatorDoor;
13     private Bell bell;
14
15     // class constructor
16     public Elevator() {}
17
18     // class methods
19     public void ride() {}
20     public void requestElevator() {}
21     public void enterElevator() {}
22     public void exitElevator() {}
23     public void departElevator() {}
24
25     // method overriding getButton
26     public Button getButton()
27     {
29        return elevatorButton;
29     }
30
31     // method overriding getDoor
32     public Door getDoor()
33     {
34        return elevatorDoor;
35     }
36  }
```

Outline

Step 2

Implement abstract classes

(Optional) Discovering Design Patterns: Introducing Creational, Structural and Behavioral Design Patterns

- Design-patterns discussion
  - Discuss each type
    - Creational – related to the creation of objects
    - Structural -- describe common ways to organize classes and objects
    - Behavioral-- Model how objects collaborate with one another
  - Discuss importance
  - Discuss how we can use each pattern in Java
- We also introduce
  - *Concurrent* design patterns
    - Used in multithreaded systems
    - Chapter 15
  - *Architectural* patterns
    - Specify how subsystems interact with each other
    - Chapter 17

Discovering Design Patterns (cont.)

- Creational design patterns
  - Address issues related to object creation
    - e.g., prevent from creating more than one object of class
    - e.g., defer until run time what type of objects to be created
  - Consider 3D drawing program
    - User can create cylinders, spheres, cubes, etc.
    - At compile time, program does not know what shapes the user will draw
    - Based on user input, program should determine this at run time

89

Discovering Design Patterns (cont.)

- 5 creational design patterns
  - Abstract Factory (Chapter 17)
  - Builder (not discussed)
  - Factory Method (Chapter 13)
  - Prototype (Chapter 21)
  - Singleton (Chapter 9)
- Singleton
  - Used when system should contain *exactly* one object of class

    Ex. one object manages database connections➔ensure that other objects cannot utilize unnecessary connections that would slow the system
  - Ensures system instantiates *maximum* of one class object

90

```
1   // Singleton.java
2   // Demonstrates Singleton design pattern
3   package com.deitel.jhtp4.designpatterns;
4
5   public final class Singleton {
6
7      // Singleton object returned by method getSingletonInstance
8      private static Singleton singleton;
9
10     // constructor prevents instantiation from other objects
11     private Singleton()
12     {
13        System.err.println( "Singleton object created." );
14     }
15
16     // create Singleton and ensure only one Singleton instance
17     public static Singleton getSingletonInstance()
18     {
19        // instantiate Singleton if null
20        if ( singleton == null )
21           singleton = new Singleton();
22
23        return singleton;
24     }
25  }
```

Outline

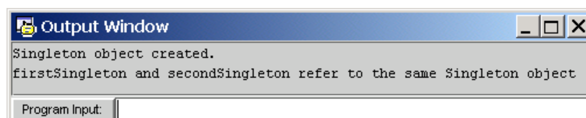
Singleton.java

Line 11
private constructor ensures only class **Singleton** can instantiate **Singleton** object

Lines 20-23
Instantiate **Singleton** object only once, but return same reference

91

```
1   // SingletonExample.java
2   // Attempt to create two Singleton objects
3   package com.deitel.jhtp4.designpatterns;
4
5   public class SingletonExample {
6
7      // run SingletonExample
8      public static void main( String args[] )
9      {
10        Singleton firstSingleton;
11        Singleton secondSingleton;
12
13        // create Singleton objects
14        firstSingleton = Singleton.getSingletonInstance();
15        secondSingleton = Singleton.getSingletonInstance();
16
17        // the "two" Singletons should refer to same Singleton
18        if ( firstSingleton == secondSingleton )
19           System.out.println( "firstSingleton and " +
20              "secondSingleton refer to the same Singleton " +
21              "object" );
22     }
23  }
```



Outline

SingletonExample.java

Line 14
Create **Singleton** instance

Line 15
Get same **Singleton** instance

92

Discovering Design Patterns (cont.)

- Structural design patterns
  - Describe common ways to organize classes and objects

  - Adapter (Chapter 13)
  - Bridge (Chapter 13)
  - Composite (Chapter 13)
  - Decorator (Chapter 17)
  - Facade (Chapter 17)
  - Flyweight (not discussed)
  - Proxy (Chapter 9)

93

Discovering Design Patterns (cont.)

- Proxy
  - Allows system to use one object instead of another
    - If original object cannot be used (for whatever reason)
  - Consider loading several large images in Java applet
    - Ideally, we want to see these image instantaneously
    - Loading these images can take time to complete
    - Applet can use gauge object that informs use of load status(what percentage of a large image has been loaded)
      - Gauge object is called the *proxy object*
    - Remove proxy object when images have finished loading→ the applet can display an image instead of the proxy object

94

Discovering Design Patterns (cont.)

- Behavioral design patterns
  - Model how objects collaborate with one another
  - Assign responsibilities to algorithms→offer special behavior appropriate for a wide variety of application

Discovering Design Patterns (cont.)

- Behavioral design patterns
  - Chain-of-Responsibility (Chapter 13)
  - Command (Chapter 13)
  - Interpreter (not discussed)
  - Iterator (Chapter 21)
  - Mediator (not discussed)
  - Memento (Chapter 9)
  - Observer (Chapter 13)
  - State (Chapter 9)
  - Strategy (Chapter 13)
  - Template Method (Chapter 13)
  - Visitor (not discussed)

Discovering Design Patterns (cont.)

- Memento
  - Allows object to save its *state* (set of attribute values)
  - Consider painting program for creating graphics
    - Offer "undo" feature if user makes mistake
      - Returns program to previous state (before error)
    - *History* lists previous program states(store several states in a list)
  - *Originator object* occupies state
    - e.g., drawing area
  - *Memento object* stores copy of originator object's attributes
    - e.g., memento saves state of drawing area
  - *Caretaker object* (history) contains references to all mementos(include originator object)
    - e.g., history lists mementos from which user can select

97

Discovering Design Patterns (cont.)

- State
  - Convey an object's state or represent the various states that an object can occupy
  - Consider optional elevator-simulation case study
    - Person walks across a floor and rides the elevator to the other floor
      - Use integer to represent floor on which person walks
    - Person rides elevator to other floor
    - On what floor is the person when riding elevator?

98

Discovering Design Patterns (cont.)

- State
  - We implement a solution:
    - Abstract superclass **`Location`**
    - Classes **`Floor`** and **`Elevator`** extend **`Location`**
    - Encapsulates information about person location
      - Each location has reference to **`Button`** and **`Door`**
    - Class **`Person`** contains **`Location`** reference
      - Reference **`Floor`** when on floor
      - Reference **`Elevator`** when in elevator