Chapter 9 - Object-Oriented Programming

Outline

© 2002 Prentice Hall. All rights reserved.

Chapter 9 - Object-Oriented Programming

© 2002 Prentice Hall. All rights reserved.

9.1 Introduction

- Object-oriented programming
  - Inheritance
    - Software reusability
    - Classes are created from existing ones
      - Absorbing attributes and behaviors
      - Adding new capabilities
      - **`Convertible`** inherits from **`Automobile`**
  - Polymorphism
    - Enables developers to write programs in general fashion
      - Handle variety of existing and yet-to-be-specified classes
    - Helps add new capabilities to system

© 2002 Prentice Hall. All rights reserved.

9.1 Introduction (cont.)

- Object-oriented programming
  - Inheritance
    - *Subclass* inherits from *superclass*
      - Subclass usually adds instance variables and methods
    - Single vs. multiple inheritance
      - Java does not support multiple inheritance
        - Interfaces (discussed later) achieve the same effect
    - "Is a" relationship
  - Composition
    - "Has a" relationship

© 2002 Prentice Hall. All rights reserved.

9.2 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

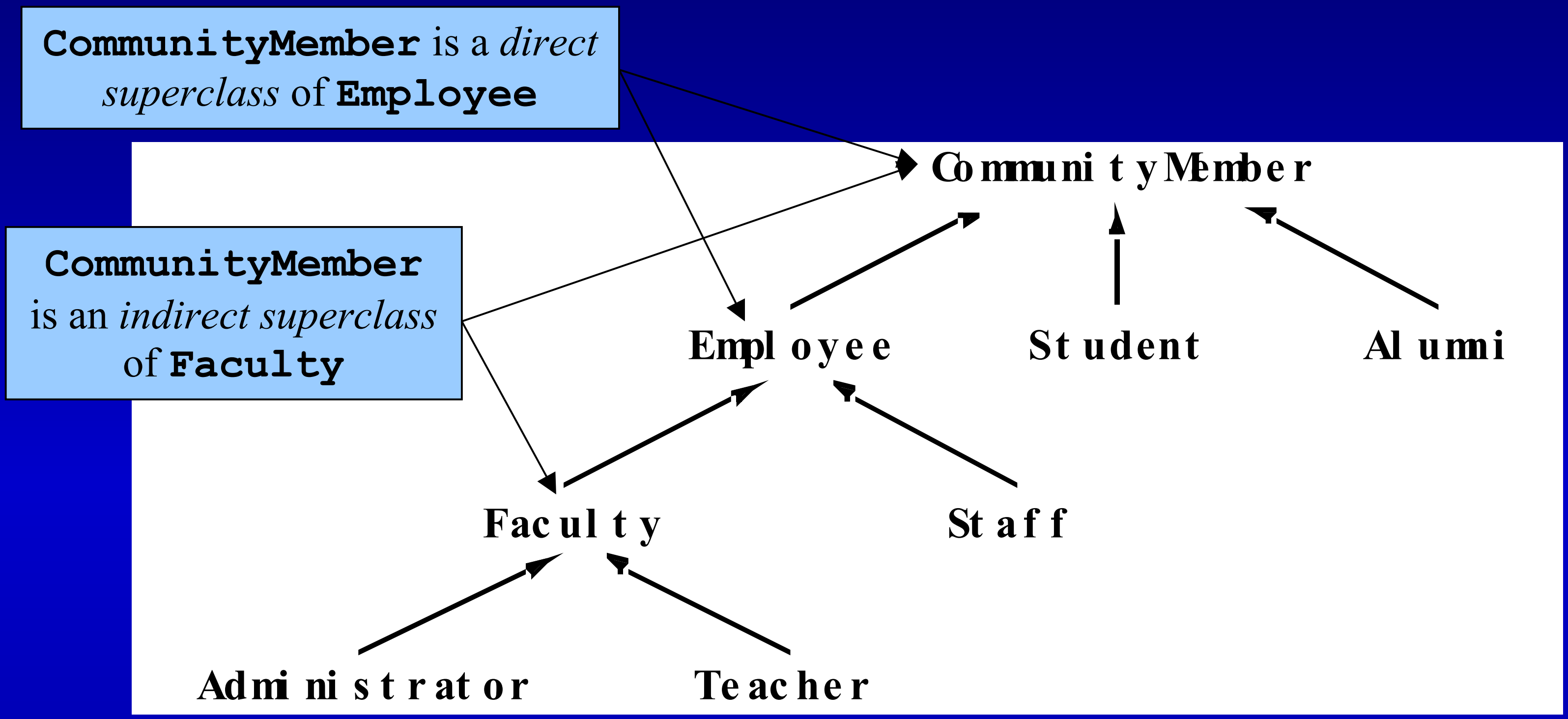
© 2002 Prentice Hall. All rights reserved.

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.

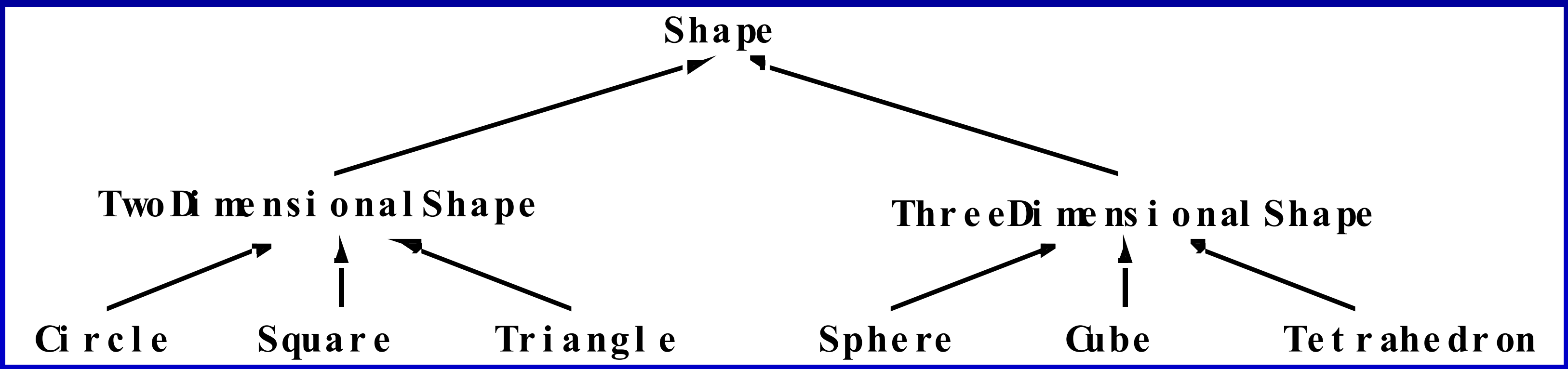
© 2002 Prentice Hall. All rights reserved.

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



© 2002 Prentice Hall. All rights reserved.

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



© 2002 Prentice Hall. All rights reserved.

9.3 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

© 2002 Prentice Hall. All rights reserved.

9.4 Relationship between Superclass Objects 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

© 2002 Prentice Hall. All rights reserved.

```
1    // Fig. 9.4: Point.java
2    // Definition of class Point
3
4    public class Point {
5       protected int x, y; // coordinates
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
```

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

Outline

`Point.java`

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

```
   // get y coordinate
   public int getY()
   {
      return y;
   }

   // convert into a String representation
   public String toString()
   {
      return "[" + x + ", " + y + "]";
   }

}  // end class Point
```

Outline

Point.java

Outline

Circle.java

```
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
```

Circle is a **Point** subclass

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

Implicit call to **Point** constructor

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

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**

```
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
```

Override method **toString** of class **Point** by using same signature

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 Circ
25         // with a Point reference
26         point2 = circle1;  // assigns Circle to a Point reference
27
28         output += "\n\nCircle circle1 (via point2 r
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
```

Instantiate **Point** and **Circle** objects

Circle invokes its overridden **toString** method

Superclass object can reference subclass object

Point still invokes **Circle**'s overridden **toString** method

Downcast **Point** to **Circle**

Outline

InheritanceTest.java

Lines 18-19
Instantiate objects

Line 22
...**cle** invokes method **toString**

Line 26
... object ... subclass

Line 29
Point invokes **Circle**'s ...**ing** method ...orphism and ...c binding)

...**oint** to

```
35          output += "\n\nCircle circle1 (vi
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    }  // end class InheritanceTest
```

Circle invokes its overridden **toString** method

Circle invokes method **area**

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

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

Outline

InheritanceTest.java

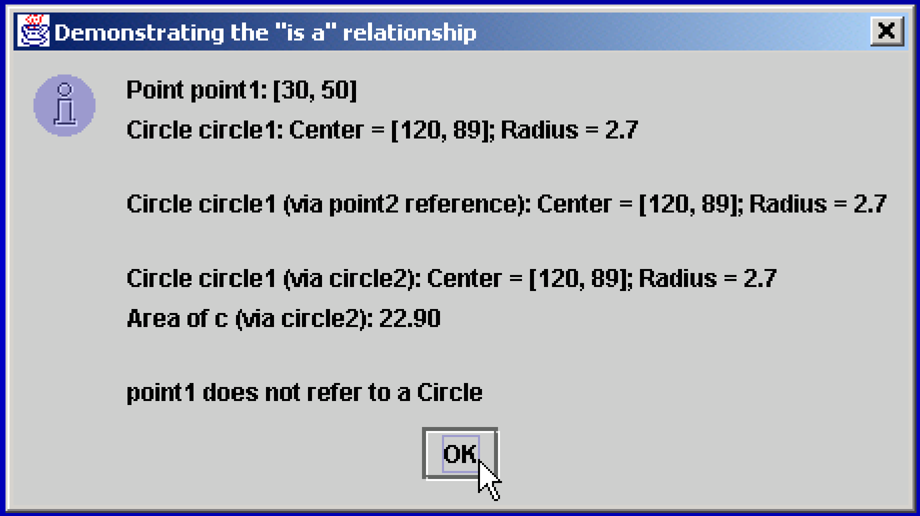
Line 36
Circle invokes its ... **toString**

Line 40
... okes ... a

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

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

Fig. 9.6 Assigning subclass references to superclass references



© 2002 Prentice Hall. All rights reserved.

9.5 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

© 2002 Prentice Hall. All rights reserved.

```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
```

Superclass constructors

Superclass **finalize** method uses **protected** for subclass access, but not for other clients

Outline

Point.java

Lines 7-20
Superclass constructors

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

Outline

`Circle.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
```

Implicit call to **`Point`** constructor

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

Override **`Point`**'s method **`finalize`**, but call it using **`super`**

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

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

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

```java
    // convert the Circle to a String
    public String toString()
    {
       return "Center = " + super.toString() +
              "; Radius = " + radius;
    }

 }  // end class Circle
```


Circle.java

```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
```

Instantiate **`Circle`** objects

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

```
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
```

Outline

`Test.java`

Lines 10-11
Instantiate **`Circle`** objects

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

9.6 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
    - Refer to superclass object with subclass reference
      - Syntax error

© 2002 Prentice Hall. All rights reserved.

9.7 Software Engineering with Inheritance

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

© 2002 Prentice Hall. All rights reserved.

9.8 Composition vs. Inheritance

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

© 2002 Prentice Hall. All rights reserved.

9.9 Case Study: Point, Cylinder, Circle

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

© 2002 Prentice Hall. All rights reserved.

```java
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
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
```

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

Constructor and overloaded constructor

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

```java
    // get y coordinate
    public int getY()
    {
       return y;
    }

    // convert into a String representation
    public String toString()
    {
       return "[" + x + ", " + y + "]";
    }

 }  // 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
```

Instantiate **Point** object

Methods **getX** and **getY** read **Point**'s **protected** variables

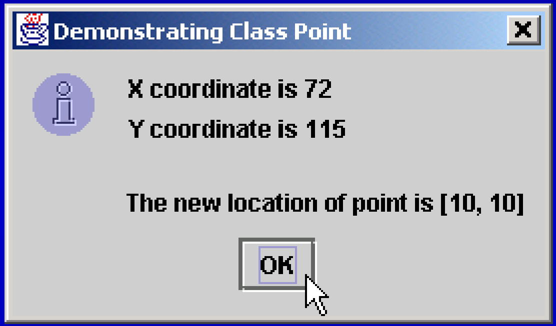
Outline

Test.java

Line 15
Instantiate **Point** object

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

Fig. 9.11 Testing class **`Point`**



© 2002 Prentice Hall. All rights reserved.

```java
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
```

Circle is a **Point** subclass

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

Implicit call to **Point** constructor

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

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**

```java
   // get radius of Circle
   public double getRadius()
   {
      return radius;
   }

   // calculate area of Circle
   public double area()
   {
      return Math.PI * radius * radius;
   }

   // convert the Circle to a String
   public String toString()
   {
      return "Center = " + "[" + x + ", " + y + "]" +
             "; Radius = " + radius;
   }

}  // 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(
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
```

Instantiate **Circle** object

Calls to methods **setRadius** read and manipulate **Circle**'s **protected**

Calls to methods **getX** **setPoint** read and manipulate inherited **protected**

Outline

Test.java

Line 19
Instantiate **Circle** object

Lines 25 and 28
Calls to methods **getRadius** and **setRadius** read and ...**e**'s ...bles

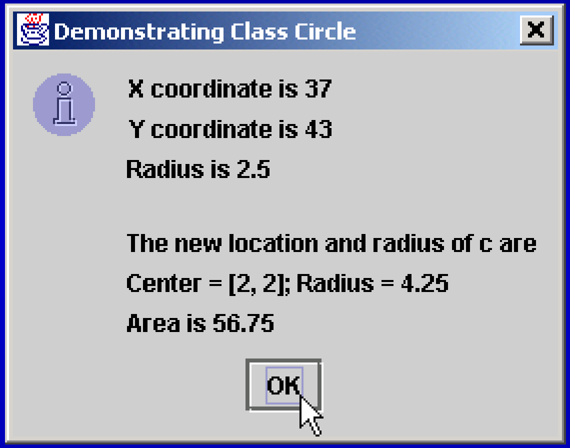
Calls to methods **getX**, **getY** and **setPoint** read and manipulate ...les

Outline

Test.java

```
         JOptionPane.showMessageDialog( null, output,
            "Demonstrating Class Circle",
            JOptionPane.INFORMATION_MESSAGE );

         System.exit( 0 );
      }

   }  // end class Test
```

Fig. 9.13 Testing class **Circle**



© 2002 Prentice Hall. All rights reserved.

```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
7
8      // no-argument constructor
9      public Cylinder()
10     {
11        // implicit call to superclass con
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
```

Cylinder is a **Circle** subclass

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

Implicit call to **Circle** constructor

Explicit call to **Circle** constructor using **super**

Outline

Cylinder.java

ne 5
linder is a
rcle subclass

Line 5
Cylinder inherits
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**

```
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
```

Override method area of class **`Circle`**

Outline

`Cylinder.java`

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

```java
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
```

Instantiate **Cylinder** object

Method calls read and manipulate **Cylinder**'s **protected** variables and inherited **protected** variables

Outline

Test.java

Line 19
Instantiate **Cylinder** object

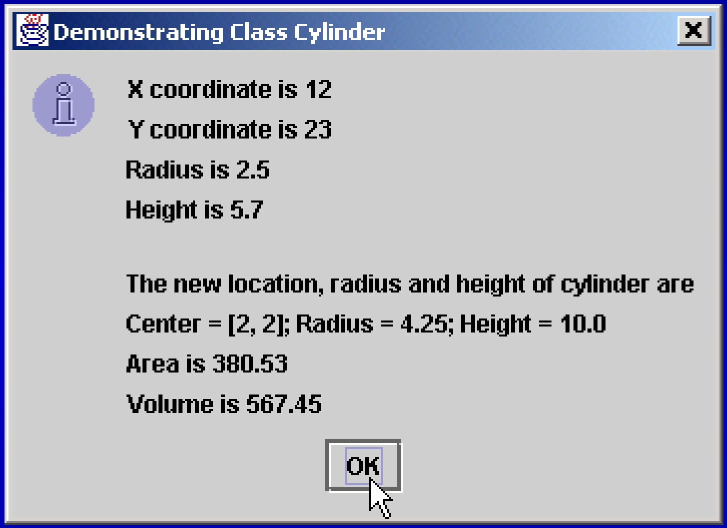
Lines 23-31
Method calls read and manipulate **Cylinder**'s **protected** variables and inherited **protected** variables

```java
         // get String representation of Cylinder and calculate
         // area and volume
         output += "\n\nThe new location, radius " +
            "and height of cylinder are\n" + cylinder +
            "\nArea is " + precision2.format( cylinder.area() ) +
            "\nVolume is " + precision2.format( cylinder.volume() );

         JOptionPane.showMessageDialog( null, output,
            "Demonstrating Class Cylinder",
            JOptionPane.INFORMATION_MESSAGE );

         System.exit( 0 );
      }

   }  // end class Test
```

Outline

Test.java

Fig. 9.15 Testing class **Test**



© 2002 Prentice Hall. All rights reserved.

9.10 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

© 2002 Prentice Hall. All rights reserved.

9.11 Type Fields and Switch Statements

- One way to deal with objects of many different types is to use a **`switch`**-based system
  - Determine appropriate action for object
    - Based on object's type
  - Error prone
    - Programmer can forget to make appropriate type test
    - Adding and deleting **`switch`** statements

© 2002 Prentice Hall. All rights reserved.

9.12 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
  - Program takes on a simplified appearance; less branching logic more sequential code
    - Facilitates testing, debugging and program maintenance

© 2002 Prentice Hall. All rights reserved.

9.12 Dynamic Method Binding (cont.)

- 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

© 2002 Prentice Hall. All rights reserved.

9.13 final Methods and Classes

- **`final`** method
  - Cannot be overridden in subclass
  - Compiler can optimize the program by removing calls to **`final`** methods and replacing them with the expanded code at each method call location – inlining the code
- **`final`** class
  - Cannot be superclass (cannot be **`extend`**ed)
    - A class cannot inherit from a **`final`** classes
    - Every method is implicitly final

© 2002 Prentice Hall. All rights reserved.

9.14 Abstract Superclasses and Concrete Classes

- Abstract classes
  - Objects cannot be instantiated
  - Too generic to define real objects
    - **`TwoDimensionalShape`**
  - Provides superclass from which other classes may inherit
    - Normally referred to as *abstract superclasses*
- Concrete classes
  - Classes from which objects are instantiated
  - Provide specifics for instantiating objects
    - **`Square`**, **`Circle`** and **`Triangle`**

© 2002 Prentice Hall. All rights reserved.

9.15 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`**

© 2002 Prentice Hall. All rights reserved.

9.16 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`**
  - Concrete **`Employee`** subclasses declared **`final`**
    - **`Boss`**
    - **`CommissionWorker`**
    - **`PieceWorker`**
    - **`HourlyWorker`**

© 2002 Prentice Hall. All rights reserved.

```
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      }
31
```

abstract class cannot be instantiated

abstract class can have instance data and non**abstract** methods for subclasses

abstract class can have constructors for subclasses to initialize inherited data

Outline

Employee.java

...ract class cannot be instantiated

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

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

```java
32     // Abstract method that must be implemented for each
33     // derived class of Employee from which objects
34     // are instantiated.
35     public abstract double earnings();
36
37  }  // end class Employee
```

Subclasses must implement **`abstract`** method

Outline

`Employee.java`

Line 35
Subclasses must implement **`abstract`** method

```
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
```

Boss is an **Employee** subclass

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

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

Required to implement **Employee**'s method **earnings** (polymorphism)

Outline

Boss.java

Boss is an **Employee** subclass

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

10
icit call to **Employee** constructor using **super**

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

```
1   // Fig. 9.18: CommissionWorker.java
2   // CommissionWorker class derived from
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
```

CommissionWorker is an **Employee** subclass

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

Outline

CommissionWorker.java

Line 4
CommissionWorker is an **Employee** subclass

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

```
31      // set CommissionWorker's quantity sold
32      public void setQuantity( in
33      {
34         quantity = ( totalSold >
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
```

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

Outline

CommissionWorker.java

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

Outline

PieceWorker.java

```
1   // Fig. 9.19: PieceWorker.java
2   // PieceWorker class derived from Emplo
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
```

PieceWorker is an **Employee** subclass

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

Implementation of **Employee**'s method **earnings**; differs from that of **Boss** and **CommissionWorker**

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**

Outline

PieceWorker.java

```
   public String toString()
   {
      return "Piece worker: " + super.toString();
   }

}  // end class PieceWorker
```

```
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
```

HourlyWorker is an **Employee** subclass

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

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

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
...s from
...**oyee**

```java
   public String toString()
   {
      return "Hourly worker: " + super.toString();
   }

}  // end class HourlyWorker
```

Outline

HourlyWorker.java

```java
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
```

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

Instantiate one instance each of **Employee** subclasses

<u>Outline</u>

Test.java

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

Lines 18-28
stance
ee
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
```

Use **Employee** to reference **Boss**

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

Do same for **CommissionWorker** and **PieceWorker**

Outline

Test.java

Line 33

to

Line 36

Method **employee.earnings** dynamically binds to

Lines 41-55

Do same for **CommissionWorker** and **PieceWorker**

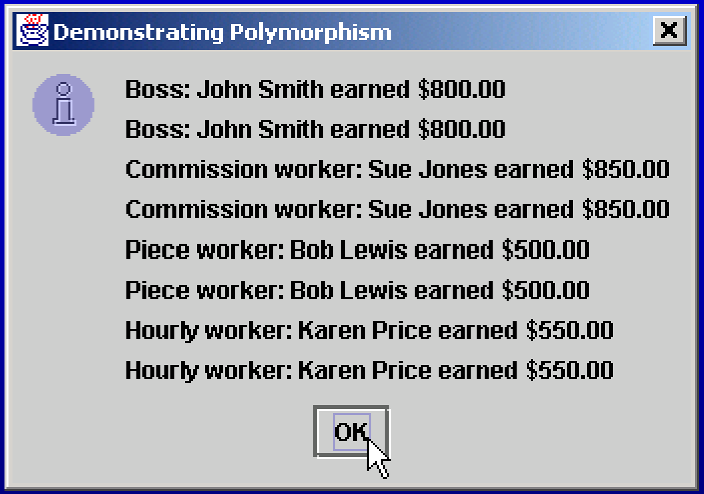
```
57       // Employee reference to an HourlyWorker
58       employee = hourlyWorker;
59
60       output += employee.toString() + " earned $" +
61       precision2.format( employee.earnings() ) + "\n" +
62            hourlyWorker.toString() + " earned $" +
63            precision2.format( hourlyWorker.earnings() ) + "\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
```

Repeat for **HourlyWorker**

Outline

Test.java

Lines 58-63
Repeat for **HourlyWorker**



9.17 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

© 2002 Prentice Hall. All rights reserved.

9.18 Case Study: Inheriting Interface and Implementation

- **`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

© 2002 Prentice Hall. All rights reserved.

```java
1   // Fig. 9.22: Shape.java
2   // Definition of abstract base class S
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` cannot be instantiated

`abstract` class can have non**`abstract`** methods for subclasses

Concrete subclasses must implement method **`getName`**

Outline

`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`**

```java
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
```

`Point` inherits **`Shape`**'s **`public`** methods

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

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
```

Implementation of **`Shape`**'s method **`getName`**

`Point` does not override methods **`area`** and **`volume`**, because points have neither area nor volume

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
```

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

Methods for reading/setting **protected** value

Outline

Circle.java

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

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

```java
36      // calculate area of Circle
37      public double area()
38      {
39         return Math.PI * radius * radius;
40      }
41
42      // convert Circle to a Stri
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
```

Override method **area** but not method **volume** (circles do not have volume)

Implementation of **Shape**’s method **getName**

Outline

Circle.java

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

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

Outline

Cylinder.java

```
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
```

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

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
```

Override methods **area** and **volume**

Implementation of **Shape**'s method **getName**

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
```

Instantiate one instance each of **Shape** subclasses

Shapes to ubclass object

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

Outline

Test.java

Line 43

bind
Name

Circle and
Cylinder objects

Line 48

d
for
Cylinder object

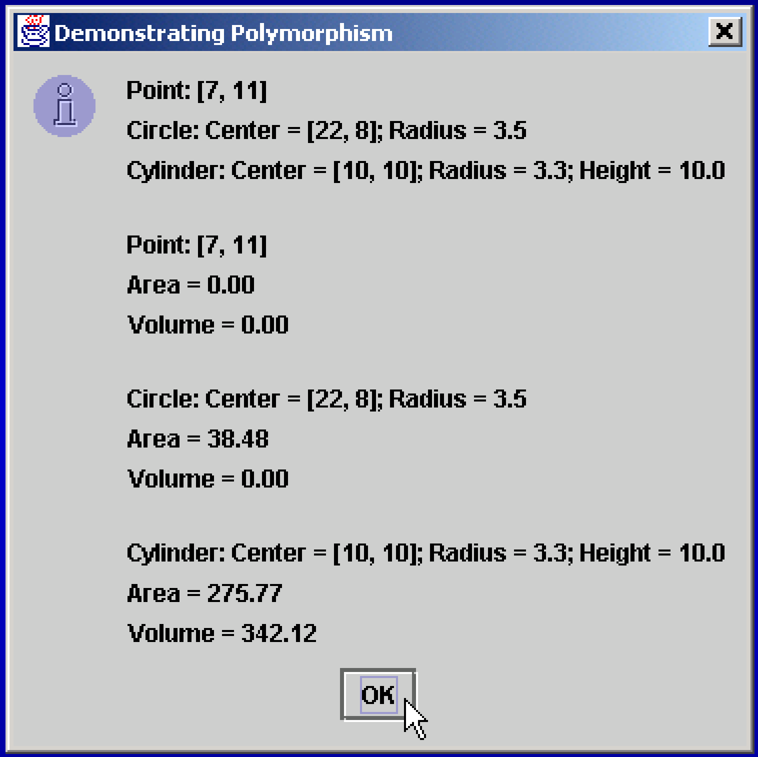
```java
32          // get name and String representation of each shape
33          String output =
34             point.getName() + ": " + point.toString() + "\n" +
35             circle.getName() + ": " + circle.toString() + "\n" +
36             cylinder.getName() + ": " + cylinder.toString();
37
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
```

Dynamically bind method **getName**

Dynamically bind method **area** for **Circle** and **Cylinder** objects

Dynamically bind method **volume** for **Cylinder** object

Test.java


Demonstrating Polymorphism
Point: [7, 11]
Circle: Center = [22, 8]; Radius = 3.5
Cylinder: Center = [10, 10]; Radius = 3.3; Height = 10.0
Point: [7, 11]
Area = 0.00
Volume = 0.00
Circle: Center = [22, 8]; Radius = 3.5
Area = 38.48
Volume = 0.00
Cylinder: Center = [10, 10]; Radius = 3.3; Height = 10.0
Area = 275.77
Volume = 342.12
OK

9.19 Case Study: Creating and Using Interfaces

- Use **`interface Shape`**
  - **`interface`** Used instead of an **`abstract`** class when there are no instance variables or default method implementations to inherit
- Interface
  - Definition begins with **`interface`** keyword
  - Classes **`implement`** an interface
  - Class must define every method in the interface with the number of arguments and return type specified in the interface
    - If any methods are undefined, the class is abstract and ust be declared so
  - Contains **`public abstract`** methods
    - Classes (that **`implement`** the interface) must implement these methods
  - A class can implement more than one interface

© 2002 Prentice Hall. All rights reserved.

```
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  }
```

Classes that **`implement Shape`** must implement these methods

Outline

`Shape.java`

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

```java
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
```

`Point` implements interface **`Shape`**

Outline

`Point.java`

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

```java
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
```

Implement methods specified by interface **Shape**

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 c
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
```

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

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
```

Override method **`toString`**

Override methods **`area`** and **`getName`** but not method **`volume`**

Outline

`Circle.java`

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

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

```java
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      }
```

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

Outline

Cylinder.java

Line 4

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

```java
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
```

Override method **toString**

Override methods **area**, **volume** and **getName**

Outline

Cylinder.java

Lines 50-53
Override method
toString

L... -59
...le methods
..., **volume** and
getName

```java
// Fig. 9.31: Test.java
// Test Point, Circle, Cylinder hierarchy with interface Shape.

// Java core packages
import java.text.DecimalFormat;

// Java extension packages
import javax.swing.JOptionPane;

public class Test {

   // test Shape hierarchy
   public static void main( String args[] )
   {
      // create shapes
      Point point = new Point( 7, 11 );
      Circle circle = new Circle( 3.5, 22, 8 );
      Cylinder cylinder = new Cylinder( 10, 3.3, 10, 10 );

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

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

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

      // aim arrayOfShapes[ 2 ] at subclass Cylinder object
      arrayOfShapes[ 2 ] = cylinder;

```

Fig. 9.31 is identical to Fig. 9.26

Outline

Test.java

Fig. 9.31 is identical to Fig. 9.26

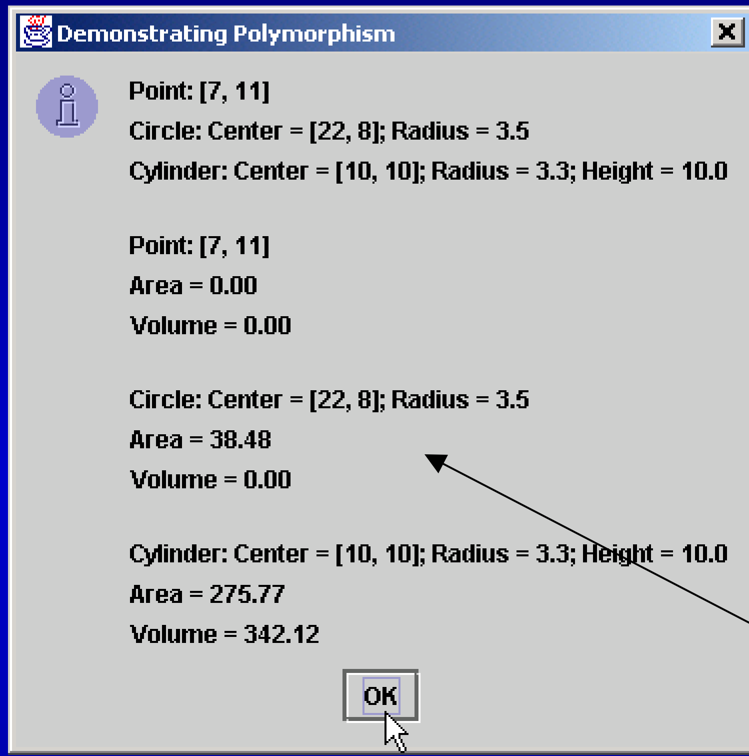
Test.java

```
      // get name and String representation of each shape
      String output =
         point.getName() + ": " + point.toString() + "\n" +
         circle.getName() + ": " + circle.toString() + "\n" +
         cylinder.getName() + ": " + cylinder.toString();

      DecimalFormat precision2 = new DecimalFormat( "0.00" );

      // loop through arrayOfShapes and get name,
      // area and volume of each shape in arrayOfShapes
      for ( int i = 0; i < arrayOfShapes.length; i++ ) {
         output += "\n\n" + arrayOfShapes[ i ].getName() +
            ": " + arrayOfShapes[ i ].toString() +
            "\nArea = " +
            precision2.format( arrayOfShapes[ i ].area() ) +
            "\nVolume = " +
            precision2.format( arrayOfShapes[ i ].volume() );
      }

      JOptionPane.showMessageDialog( null, output,
         "Demonstrating Polymorphism",
         JOptionPane.INFORMATION_MESSAGE );

      System.exit( 0 );
   }

}  // end class Test
```

Outline

Test.java

Output is identical to that of Fig. 9.26



Output is identical to that of Fig. 9.26

9.19 Interfaces

- Interfaces can also define a set of constants that can be used in many class definitions

```
public interface Constants{
   public static final int ONE=1;
   public static final int TWO=2;
   public static final int THREE=3;
}
```

- Classes that implement interface **`Constants`** can use **`ONE`**, **`TWO`** and **`THREE`** anywhere in the class definition
- Classes that import the interface can refer to them as **`Constants.ONE, Constants.TWO`**
- As there are no methods in this interface, a class that implements it is nt required to provide any method implementations

© 2002 Prentice Hall. All rights reserved.

9.20 Inner Class Definitions

- Inner classes
  - Class is defined inside another class body
  - Frequently used with GUI handling
    - Declare **`ActionListener`** inner class
    - GUI components can register **`ActionListener`**s for events
      - Button events, key events, etc.
  - An inner class is allowed to access directly all the instance variables and methods of the outer class
  - An inner class defined in a method is allowed access directly to all the instance variables and methods of the outer class object that defined it and any final local variables in the method

© 2002 Prentice Hall. All rights reserved.

```
1    // Fig. 9.32: Time.java
2    // Time class definition.
3
4    // Java core packages
5    import java.text.DecimalFormat;
6
7    // This class maintains the time in 24-hour format
8    public class Time extends Object {
9       private int hour;     // 0 - 23
10      private int minute;   // 0 - 59
11      private int second;   // 0 - 59
12
13      // Time constructor initializes each instance variable
14      // to zero. Ensures that Time object starts in a
15      // consistent state.
16      public Time()
17      {
18         setTime( 0, 0, 0 );
19      }
20
21      // Set a new time value using universal time. Perform
22      // validity checks on the data. Set invalid values to zero.
23      public void setTime( int hour, int minute, int second )
24      {
25         setHour( hour );
26         setMinute( minute );
27         setSecond( second );
28      }
29
30      // validate and set hour
31      public void setHour( int h )
32      {
33         hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
34      }
35
```

Same **Time** class used in Chapter 8

Outline

Time.java

Line 8
Same **Time** class used in Chapter 8

```java
   // validate and set minute
   public void setMinute( int m )
   {
      minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
   }

   // validate and set second
   public void setSecond( int s )
   {
      second = ( ( s >= 0 && s < 60 ) ? s : 0 );
   }

   // get hour
   public int getHour()
   {
      return hour;
   }

   // get minute
   public int getMinute()
   {
      return minute;
   }

   // get second
   public int getSecond()
   {
      return second;
   }

```

Mutator and accessor methods

Outline

`Time.java`

Mutator and accessor methods

```java
66     // convert to String in standard-time format
67     public String toString()
68     {
69        DecimalFormat twoDigits = new DecimalFormat( "00" );
70
71        return ( ( getHour() == 12 || getHour() == 0 ) ?
72           12 : getHour() % 12 ) + ":" +
73           twoDigits.format( getMinute() ) + ":" +
74           twoDigits.format( getSecond() ) +
75           ( getHour() < 12 ? " AM" : " PM" );
76     }
77
78  }  // end class Time
```

Override method **`java.lang.Object.toString`**

Outline

`Time.java`

Lines 67-76
Override method **`java.lang.Object`** **`.toString`**

```
1    // Fig. 9.33: TimeTestWindow.java
2    // Demonstrating the Time class set and get methods
3
4    // Java core packages
5    import java.awt.*;
6    import java.awt.event.*;
7
8    // Java extension packages
9    import javax.swing.*;
10
11   public class TimeTestWindow extends JFrame {
12      private Time time;
13      private JLabel hourLabel, minuteLabel, secondLabel;
14      private JTextField hourField, minuteField,
15         secondField, displayField;
16      private JButton exitButton;
17
18      // set up GUI
19      public TimeTestWindow()
20      {
21         super( "Inner Class Demonstration" );
22
23         time = new Time();
24
25         // create an instance of inner class ActionEventHandler
26         ActionEventHandler handler = new ActionEventHandler();
27
28         // set up GUI
29         Container container = getContentPane();
30         container.setLayout( new FlowLayout() );
31
```

JFrame provides basic window attributes and behaviors

JFrame (unlike **JApplet**) has constructor

Instantiate **Time** object

Instantiate object of inner-class that implements **ActionListener**

Outline

TimeTestWindow.java

Line 11
JFrame provides basic window attributes and behaviors

Line 19
me (unlike
let) has
constructor

Line 23
Instantiate **Time** object

Line 26
Instantiate object of inner-class that
istener

```
32         hourLabel = new JLabel( "Set Hour" );
33         hourField = new JTextField( 10 );
34         hourField.addActionListener( handler );
35         container.add( hourLabel );
36         container.add( hourField );
37 
38         minuteLabel = new JLabel( "Set minute" );
39         minuteField = new JTextField( 10 );
40         minuteField.addActionListener( handler );
41         container.add( minuteLabel );
42         container.add( minuteField );
43 
44         secondLabel = new JLabel( "Set Second" );
45         secondField = new JTextField( 10 );
46         secondField.addActionListener( handler );
47         container.add( secondLabel );
48         container.add( secondField );
49 
50         displayField = new JTextField( 30 );
51         displayField.setEditable( false );
52         container.add( displayField );
53 
54         exitButton = new JButton( "Exit" );
55         exitButton.addActionListener( handler );
56         container.add( exitButton );
57 
58      }  // end constructor
59 
60      // display time in displayField
61      public void displayTime()
62      {
63         displayField.setText( "The time is: " + time );
64      }
65 
```

Register **ActionEventHandler** with GUI components

Outline

TimeTestWindow.java

Line 34, 40, 46 and 55 Register **Action-EventHandler** with GUI components

```java
66      // create TimeTestWindow and display it
67      public static void main( String args[] )
68      {
69         TimeTestWindow window = new TimeTestWindow();
70
71         window.setSize( 400, 140 );
72         window.setVisible( true );
73      }
74
75      // inner class definition for handling JTextField and
76      // JButton events
77      private class ActionEventHandler
78         implements ActionListener {
79
80         // method to handle action events
81         public void actionPerformed( ActionEvent event )
82         {
83            // user pressed exitButton
84            if ( event.getSource() == ex
85               System.exit( 0 );   // te
86
87            // user pressed Enter key in hourField
88            else if ( event.getSource() == hourField ) {
89               time.setHour(
90                  Integer.parseInt( event.getActionCommand() ) );
91               hourField.setText( "" );
92            }
93
94            // user pressed Enter key in minuteField
95            else if ( event.getSource() == minuteField ) {
96               time.setMinute(
97                  Integer.parseInt( event.getActionCommand() ) );
98               minuteField.setText( "" );
99            }
```

Define inner class that implements **ActionListener** interface

Must implement method **actionPerformed** of **ActionListener**

When user presses **JButton** or **Enter** key, method **actionPerformed** is invoked

Determine action depending on where event originated

Outline

TimeTestWindow.java

Lines 77-78
Define inner class

ent
method
actionPerformed

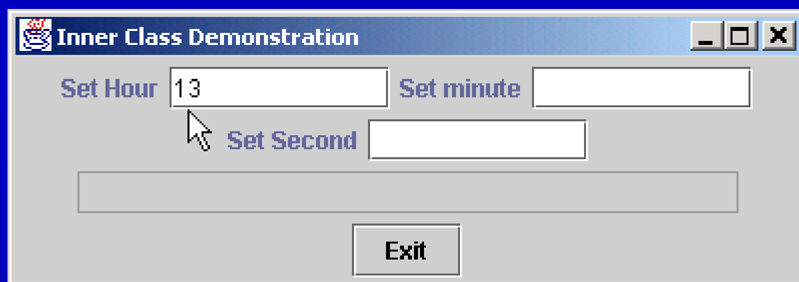
resses
, method
actionPerformed
is invoked

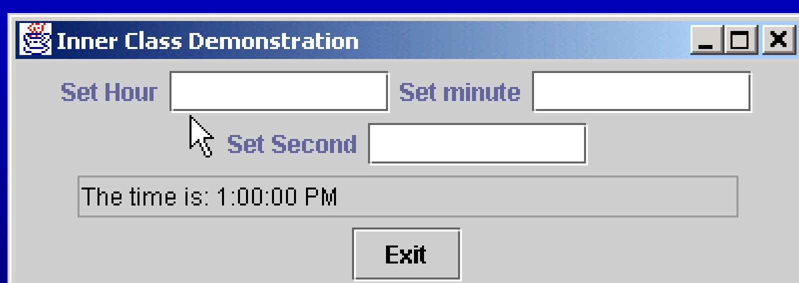
tion
where
event originated

Outline

TimeTestWindow.java

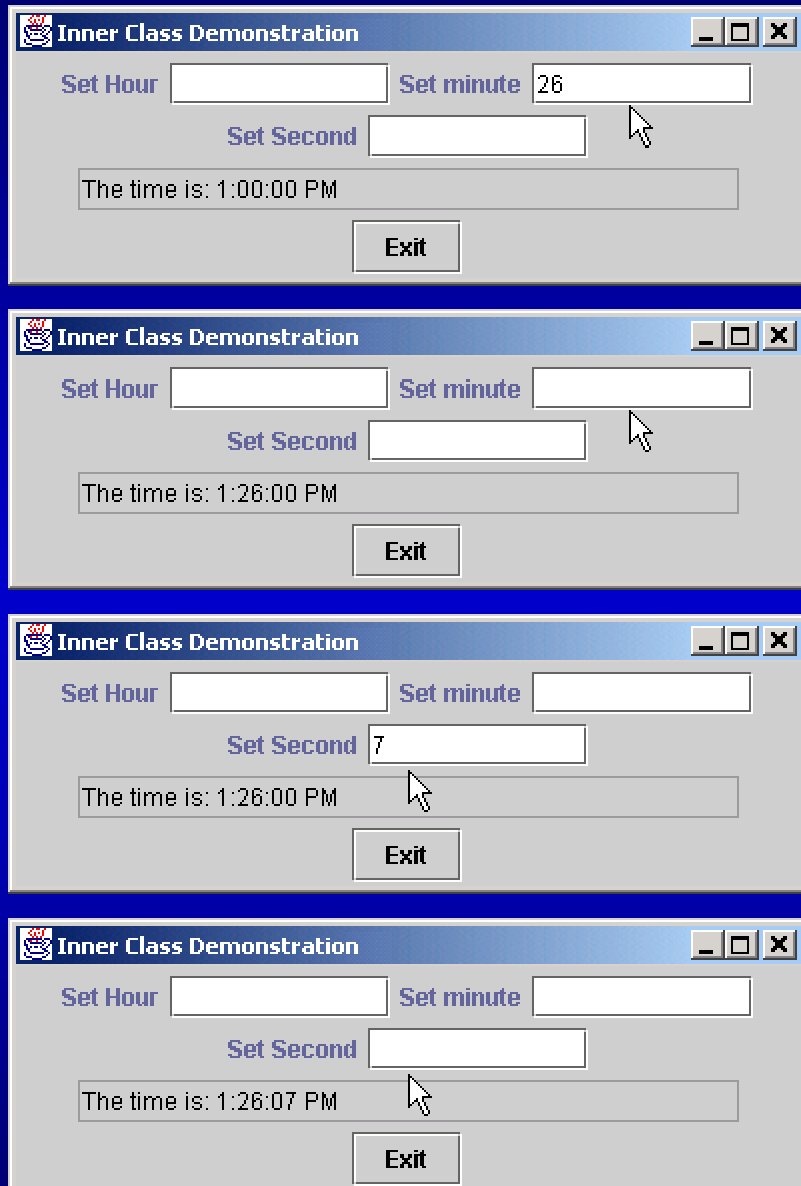
```
         // user pressed Enter key in secondField
         else if ( event.getSource() == secondField ) {
            time.setSecond(
               Integer.parseInt( event.getActionCommand() ) );
            secondField.setText( "" );
         }

         displayTime();
      }

   }  // end inner class ActionEventHandler

}  // end class TimeTestWindow
```





Outline

TimeTestWindow.java



9.20 Inner Class Definitions (cont.)

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

© 2002 Prentice Hall. All rights reserved.

```java
// Fig. 9.34: TimeTestWindow.java
// Demonstrating the Time class set and get methods

// Java core packages
import java.awt.*;
import java.awt.event.*;

// Java extension packages
import javax.swing.*;

public class TimeTestWindow extends JFrame {
   private Time time;
   private JLabel hourLabel, minuteLabel, secondLabel;
   private JTextField hourField, minuteField,
      secondField, displayField;

   // set up GUI
   public TimeTestWindow()
   {
      super( "Inner Class Demonstration" );

      // create Time object
      time = new Time();

      // create GUI
      Container container = getContentPane();
      container.setLayout( new FlowLayout() );

      hourLabel = new JLabel( "Set Hour" );
      hourField = new JTextField( 10 );

```

```
32          // register hourField event handler
33          hourField.addActionListener(
34
35             // anonymous inner class
36             new ActionListener() {
37
38                public void actionPerformed( ActionEvent event )
39                {
40                   time.setHour(
41                      Integer.parseInt( event.getActionCommand() ) );
42                   hourField.setText( "" );
43                   displayTime();
44                }
45
46             }  // end anonymous inner class
47
48          ); // end call to addActionListener
49
50          container.add( hourLabel );
51          container.add( hourField );
52
53          minuteLabel = new JLabel( "Set minute" );
54          minuteField = new JTextField( 10 );
55
56          // register minuteField event handler
57          minuteField.addActionListener(
58
59             // anonymous inner class
60             new ActionListener() {
61
```

Define anonymous inner class that implements **ActionListener**

Inner class implements method **actionPerformed** of **ActionListener**

Pass **ActionListener** as argument to GUI component's method **addActionListener**

Repeat process for **JTextField** **minuteField**

Outline

TimeTestWindow.java

Line 36
Define anonymous inner class

...4
...s implements
method
...nPerformed

Pass **Action-Listener** to GUI component's method **addAction-Listener**

Line 57-60
...t process for
...teField

```java
62              public void actionPerformed( ActionEvent event )
63              {
64                 time.setMinute(
65                    Integer.parseInt( event.getActionCommand() ) );
66                 minuteField.setText( "" );
67                 displayTime();
68              }
69
70           }  // end anonymous inner cla
71
72        ); // end call to addActionListener
73
74        container.add( minuteLabel );
75        container.add( minuteField );
76
77        secondLabel = new JLabel( "Set Second" );
78        secondField = new JTextField( 10 );
79
80        secondField.addActionListener(
81
82           // anonymous inner class
83           new ActionListener() {
84
85              public void actionPerformed( ActionEvent event )
86              {
87                 time.setSecond(
88                    Integer.parseInt( event.getActionCommand() ) );
89                 secondField.setText( "" );
90                 displayTime();
91              }
92
93           }  // end anonymous inner cla
94
95        ); // end call to addActionListener
96
```

Logic differs from logic in **actionPerformed** method of **hourField**'s inner class

Repeat process for **JTextField** **secondField**

Logic differs from logic in **actionPerformed** methods of other inner classes

Outline

TimeTestWindow.java

Line 64-67
Logic differs from logic in **action-Performed** method of **hourField**'s inner class

Line 80-83
Repeat process for **JTextField** **secondField**

Line 87-90
Logic differs from logic in **action-Performed** methods of other inner classes

```
97        container.add( secondLabel );
98        container.add( secondField );
99
100       displayField = new JTextField( 30 );
101       displayField.setEditable( false );
102       container.add( displayField );
103    }
104
105    // display time in displayField
106    public void displayTime()
107    {
108       displayField.setText( "The time is: " + time );
109    }
110
111    // create TimeTestWindow, register for its window events
112    // and display it to begin application's execution
113    public static void main( String args[] )
114    {
115       TimeTestWindow window = new TimeTestWindow();
116
117       // register listener for windowClosing event
118       window.addWindowListener(
119
120          // anonymous inner class for windowClosing event
121          new WindowAdapter() {
122
123             // terminate application when user closes window
124             public void windowClosing( WindowEvent event )
125             {
126                System.exit( 0 );
127             }
128
129          }  // end anonymous inner class
130
131      ); // end call to addWindowListener
```

Define anonymous inner class that extends **`WindowsAdapter`** to enable closing of **`JFrame`**

Outline

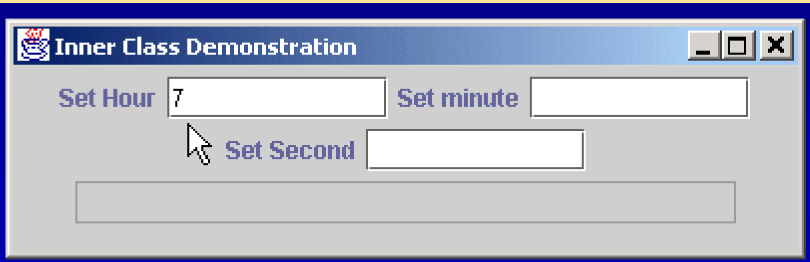
`TimeTestWindow.java`

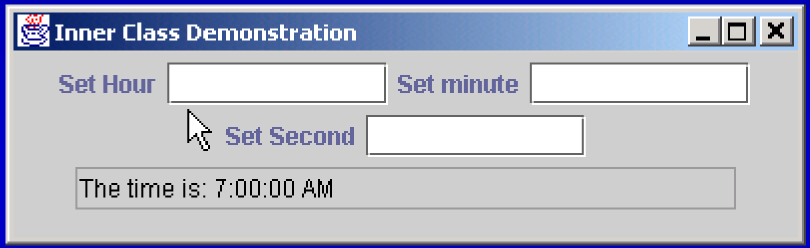
Line 118-131
Define anonymous inner class that extends **`WindowsAdapter`** to enable closing of **`JFrame`**

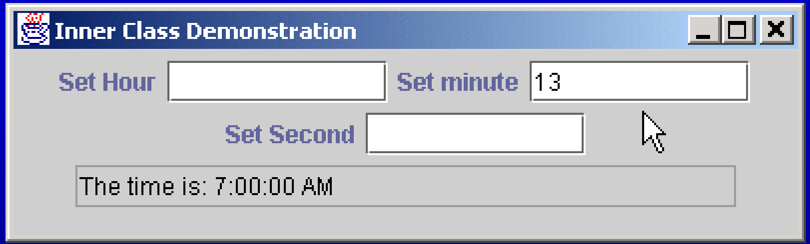
Outline

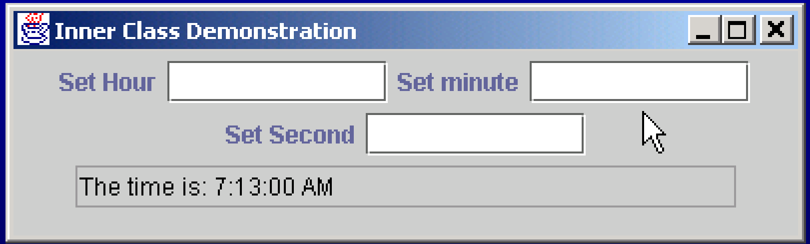
TimeTestWindow.java

```

      window.setSize( 400, 120 );
      window.setVisible( true );
   }

}  // end class TimeTestWindow
```



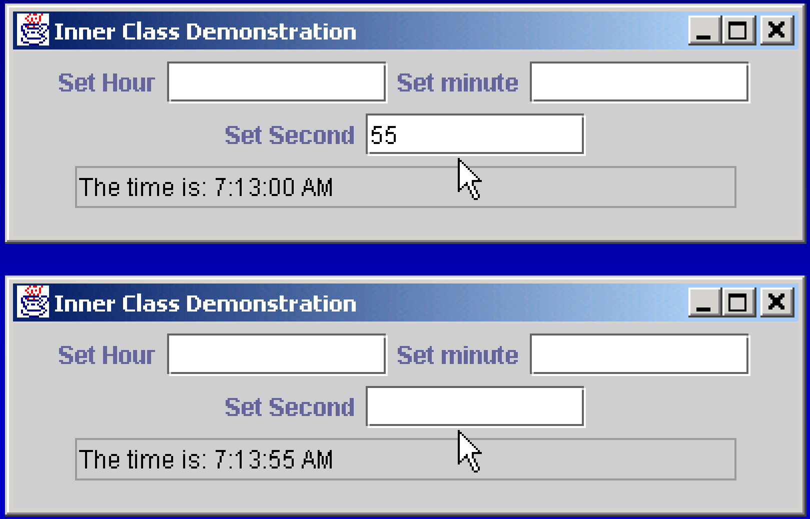






Outline

TimeTestWindow.java



9.21 Notes on Inner Class Definitions

- Notes for inner-class definition and use
  - Compiling class that contains inner class
    - Results in separate **`.class`** file
    - Inner classes with names have the file name *OuterClassName$InnerClassName*.class
    - Anonymous inner class have the file name *OuterClassName$#*.class where # starts at 1 and is incremented for each additional anonymous inner class
  - Inner classes with class names can be defined as
    - **`public`**, **`protected`**, **`private`** or package access
  - Access outer class' **`this`** reference *OuterClassName*.**`this`**
  - When an anonymous inner class implements an interface, the class must define every method in the interface
    - There are 7 different methods that must be defined in every class that implements WindowListener. We only need one (windowClosing). Java provides an adapter class that already implements all the methods of the interface. We extend the adapter class and override just the methods we need to change.

© 2002 Prentice Hall. All rights reserved.

9.22 Type-Wrapper Classes for Primitive Types

- Type-wrapper class
  - Each primitive type has one
    - **`Character`**, **`Byte`**, **`Integer`**, **`Boolean`**, etc.
    - Numeric types inherit from class **`Number`**
  - Enable to represent and manipulate primitive as objects of class **`Object`**
    - Primitive data types can be processed polymorphically
  - Declared as **`final`**
  - Many methods are declared **`static`**

© 2002 Prentice Hall. All rights reserved.

9.23 (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
    - Extract commonality into superclass
      - Subclasses inherits this commonality

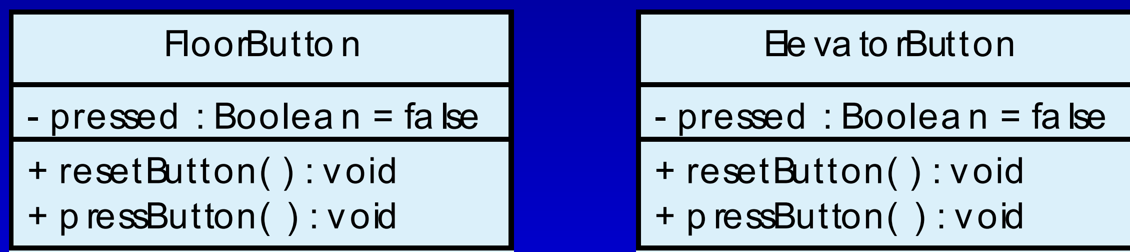
© 2002 Prentice Hall. All rights reserved.

9.23 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

© 2002 Prentice Hall. All rights reserved.

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



© 2002 Prentice Hall. All rights reserved.

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`**

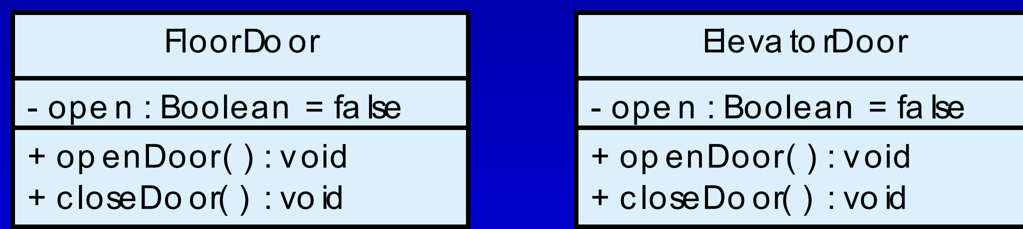
© 2002 Prentice Hall. All rights reserved.

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`**

© 2002 Prentice Hall. All rights reserved.

Fig. 9.36 Attributes and operations of classes FloorDoor and ElevatorDoor



© 2002 Prentice Hall. All rights reserved.

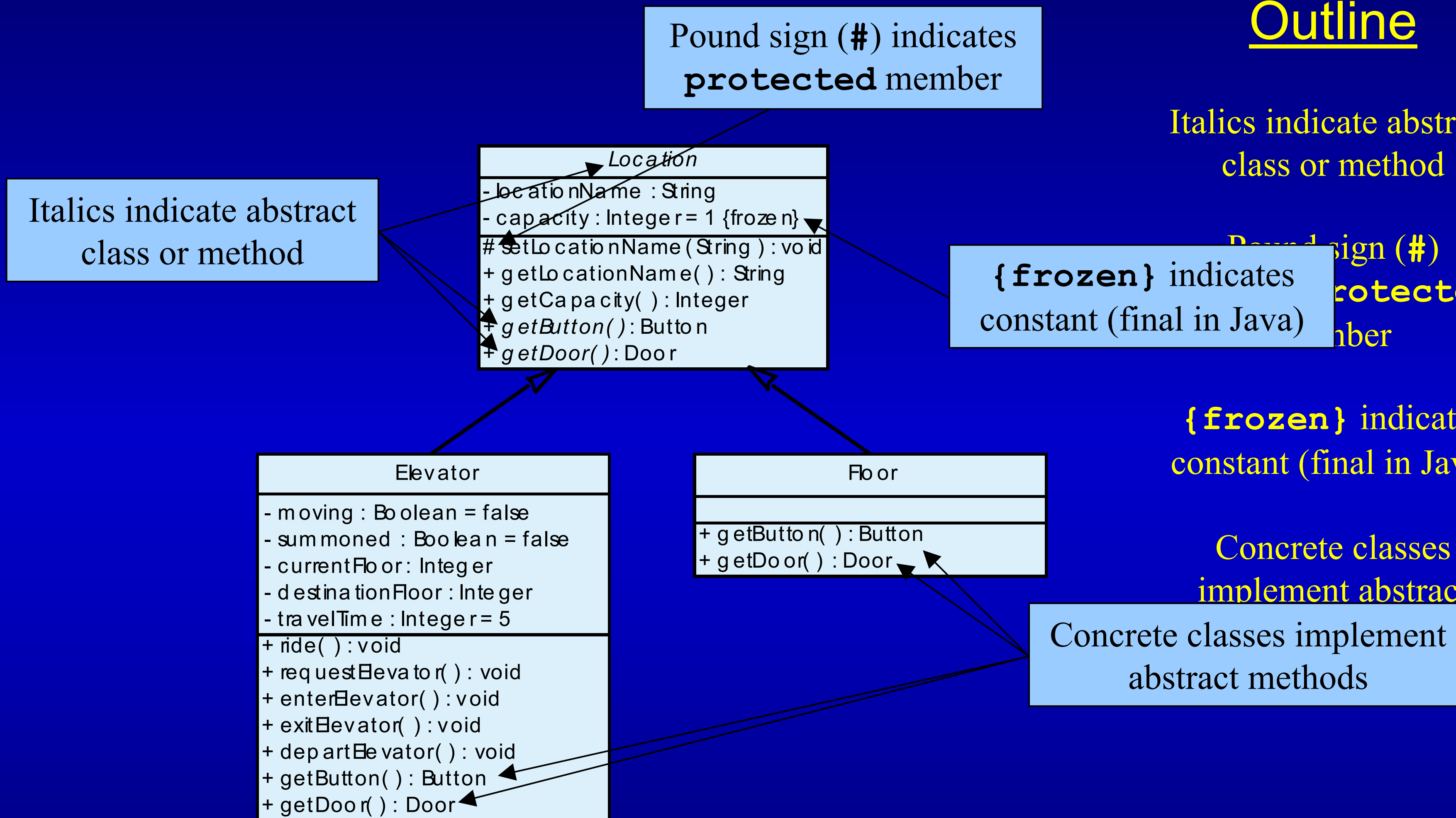
9.23 Thinking About Objects (cont.)

- 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 **`Location`** reference
    - References **`Elevator`** when person is in elevator
    - References **`Floor`** when person is on floor

© 2002 Prentice Hall. All rights reserved.

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

Outline



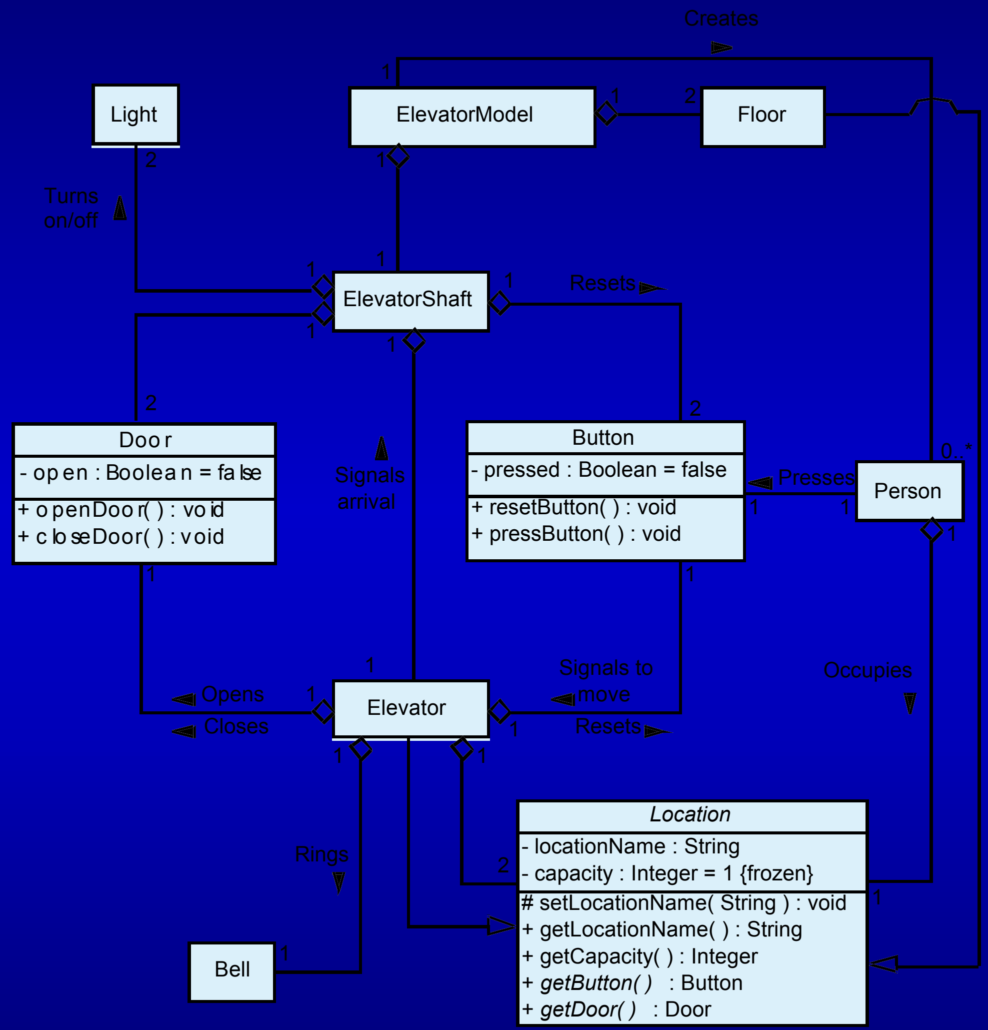
Italics indicate abstract class or method

{frozen} indicates constant (final in Java)

Concrete classes implement abstract

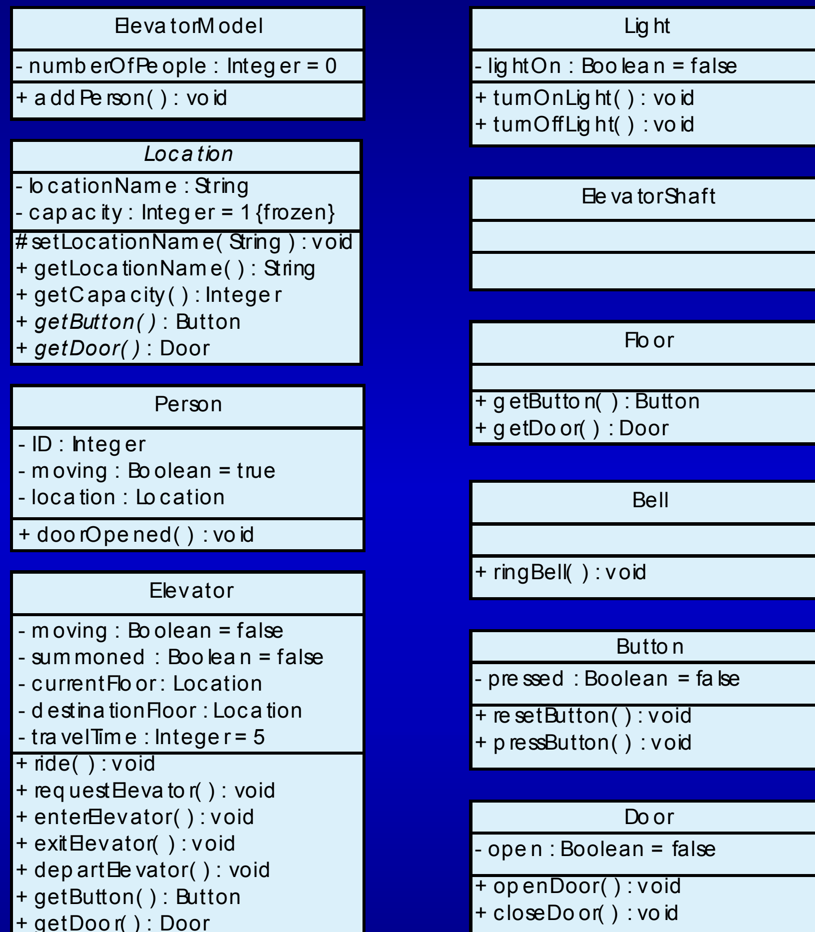
© 2002 Prentice Hall. All rights reserved.

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



© 2002 Prentice Hall. All rights reserved.

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



© 2002 Prentice Hall. All rights reserved.

9.23 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)

© 2002 Prentice Hall. All rights reserved.

9.23 Thinking About Objects (cont.)
Step 1

```
public class Elevator extends Location {

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

© 2002 Prentice Hall. All rights reserved.

```java
// Elevator.java
// Generated using class diagrams 9.38 and 9.39
public class Elevator extends Location {

   // class attributes
   private boolean moving;
   private boolean summoned;
   private Location currentFloor;
   private Location destinationFloor;
   private int travelTime = 5;
   private Button elevatorButton;
   private Door elevatorDoor;
   private Bell bell;

   // class constructor
   public Elevator() {}

   // class methods
   public void ride() {}
   public void requestElevator() {}
   public void enterElevator() {}
   public void exitElevator() {}
   public void departElevator() {}

   // method overriding getButton
   public Button getButton()
   {
      return elevatorButton;
   }

   // method overriding getDoor
   public Door getDoor()
   {
      return elevatorDoor;
   }
}
```

Implement abstract classes

Outline

Step 2

Implement abstract classes

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

- Design-patterns discussion
  - Discuss each type
    - Creational
    - Structural
    - Behavioral
  - Discuss importance
  - Discuss how we can use each pattern in Java

© 2002 Prentice Hall. All rights reserved.

9.24 Discovering Design Patterns (cont.)

- We also introduce
  - *Concurrent* design patterns
    - Used in multithreaded systems
    - Chapter 15
  - *Architectural* patterns
    - Specify how subsystems interact with each other
    - Chapter 17

© 2002 Prentice Hall. All rights reserved.

Section	Creational design patterns	Structural design patterns	Behavioral design patterns
9.24	Singleton	Proxy	Memento State
13.18	Factory Method	Adapter Bridge Composite	Chain-of-Responsibility Command Observer Strategy Template Method
17.11	Abstract Factory	Decorator Facade	
21.12	Prototype		Iterator

Fig. 9.40 The 18 Gang-of-four design patterns discussed in *Java How to Program 4/e*.

© 2002 Prentice Hall. All rights reserved.

Section	Concurrent design patterns	Architectural patterns
15.13	Single-Threaded Execution Guarded Suspension Balking Read/Write Lock Two-Phase Termination	
17.11		Model-View-Controller Layers

Fig. 9.41 Concurrent design patterns and architectural patterns discussed in *Java How to Program, 4/e*.

© 2002 Prentice Hall. All rights reserved.

9.24 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 at 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

© 2002 Prentice Hall. All rights reserved.

9.24 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)

© 2002 Prentice Hall. All rights reserved.

9.24 Discovering Design Patterns (cont.)

- Singleton
  - Used when system should contain *exactly* one object of class
    - e.g., one object manages database connections
  - Ensures system instantiates *maximum* of one class object

© 2002 Prentice Hall. All rights reserved.

```java
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 fro
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   }
```

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

Instantiate **Singleton** object only once, but return same reference

Outline

Singleton.java

Line 11
private constructor
only class
eton can
ate
Singleton object

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

Outline

SingletonExample.java

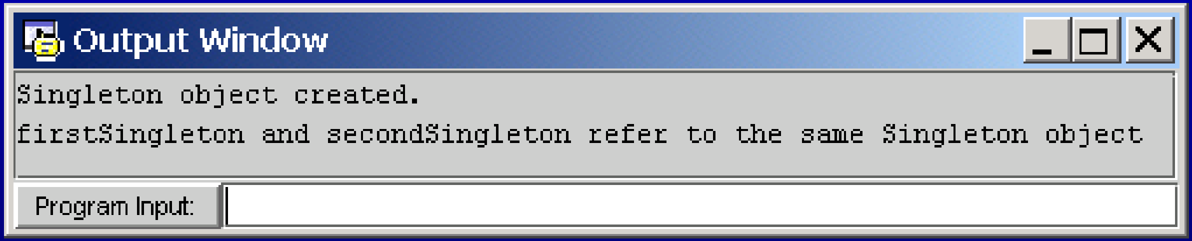
Line 14
Create **Singleton** instance

Line 15
Get same **Singleton** instance

```java
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 }
```

Create **Singleton** instance

Get same **Singleton** instance



9.24 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)

© 2002 Prentice Hall. All rights reserved.

9.24 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
      – Gauge object is called the *proxy object*
    - Remove proxy object when images have finished loading

© 2002 Prentice Hall. All rights reserved.

9.24 Discovering Design Patterns (cont.)

- Behavioral design patterns
  - Model how objects collaborate with one another
  - Assign responsibilities to algorithms

© 2002 Prentice Hall. All rights reserved.

9.24 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)

© 2002 Prentice Hall. All rights reserved.

9.24 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
  - *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 mementos
    - e.g., history lists mementos from which user can select

© 2002 Prentice Hall. All rights reserved.

9.24 Discovering Design Patterns (cont.)

- State
  - Encapsulates object's state
  - Consider optional elevator-simulation case study
    - Person walks on floor toward elevator
      - Use integer to represent floor on which person walks
    - Person rides elevator to other floor
    - On what floor is the person when riding elevator?

© 2002 Prentice Hall. All rights reserved.

9.24 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

© 2002 Prentice Hall. All rights reserved.