Advanced OOP with Java

Agenda

- Inheritance
  - Super-class and Sub-class
  - Constructors and destructors
- Polymorphism
  - Relationships among objects in an inheritance hierarchy
  - Invoking super-class methods from sub-class objects
  - Using super-class references with subclass-type variables
  - Sub-class method calls via super-class-Type variables
  - Abstract classes and methods
  - Interfaces
  - Overload and Override

Learning Approach

- The following are strongly suggested for a better learning and understanding of this course:
  - Noting down the key concepts in the class
  - Analyze all the examples / code snippets provided
  - Study and understand the self study topics
  - Completion and submission of all the assignments, on time
  - Completion of the self review questions in the lab guide
  - Study and understand all the artifacts including the reference materials / e-learning / supplementary materials specified
  - Completion of the project (if application for this course) on time inclusive of individual and group activities
  - Taking part in the self assessment activities
  - Participation in the doubt clearing sessions

Session 1

INHERITANCE

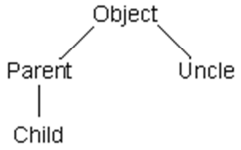
Inheritance

Inheritance allows you to define a new class by specifying only the ways in which it differs from an existing class. Inheritance promotes software reusability

- Create new class from existing class
  - Absorb existing class's data and behaviors
  - Enhance with new capabilities
- Subclass extends superclass
  - Subclass
    - More specialized group of objects
    - Behaviors inherited from superclass
      - » Can customize
    - Additional behaviors

Inheritance

- Class hierarchy
  - Direct superclass
    - Inherited explicitly (one level up hierarchy)
  - Indirect superclass
    - Inherited two or more levels up hierarchy
  - Single inheritance
    - Inherits from one superclass
  - Multiple inheritance
    - Inherits from multiple superclasses
      - Java does not support multiple inheritance in ***classes***



Java does not support multiple inheritance in classes, but interface is OK.

Inheritance

- "IS-A" vs. "HAS-A"
  - "IS-A" relationship – this thing **is a** type of that thing
    - Inheritance
    - Subclass object treated as superclass object
    - Example: Car *is a* Vehicle
      - Vehicle properties/behaviors also car properties/behaviors
  - "HAS-A" relationship: class A HAS-A B if code in class A has a reference to an instance of class B.
    - Composition
    - Object contains one or more objects of other classes as members
    - Example: Car *has a* SteeringWheel

Superclasses and Subclasses

- Superclasses and subclasses
  - Object of one class “is a” object of another class
    - Example: Rectangle is quadrilateral.
      - Class Rectangle inherits from class Quadrilateral
      - Quadrilateral: superclass
      - Rectangle: subclass
  - Superclass typically represents larger set of objects than subclasses
    - Example:
      - superclass: Vehicle
        - » Cars, trucks, boats, bicycles, …
      - subclass: Car
        - » Smaller, more-specific subset of vehicles

Final class

- You can declare an class is final - this prevents the class from being subclassed.
- Of course, an abstract class cannot be a final class.

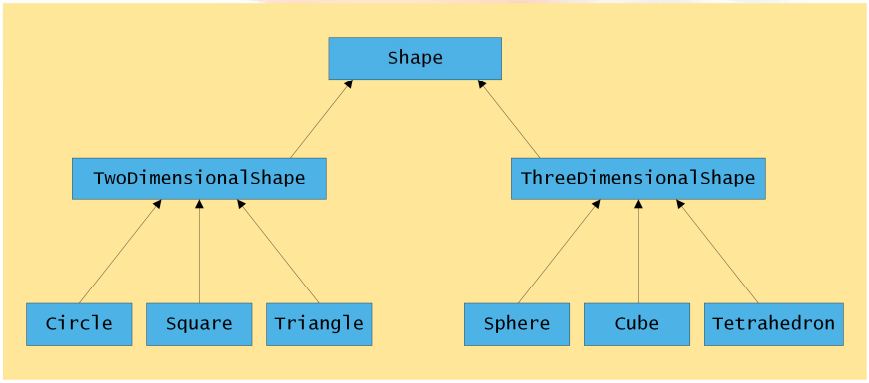
Superclasses and Subclasses (Cont.)

- Inheritance examples

Superclass	Subclasses
`Student`	`GraduateStudent`, `UndergraduateStudent`
`Shape`	`Circle`, `Triangle`, `Rectangle`
`Loan`	`CarLoan`, `HomeImprovementLoan`, `MortgageLoan`
`Employee`	`Faculty`, `Staff`
`BankAccount`	`CheckingAccount`, `SavingsAccount`

Inheritance examples.

Inheritance hierarchy



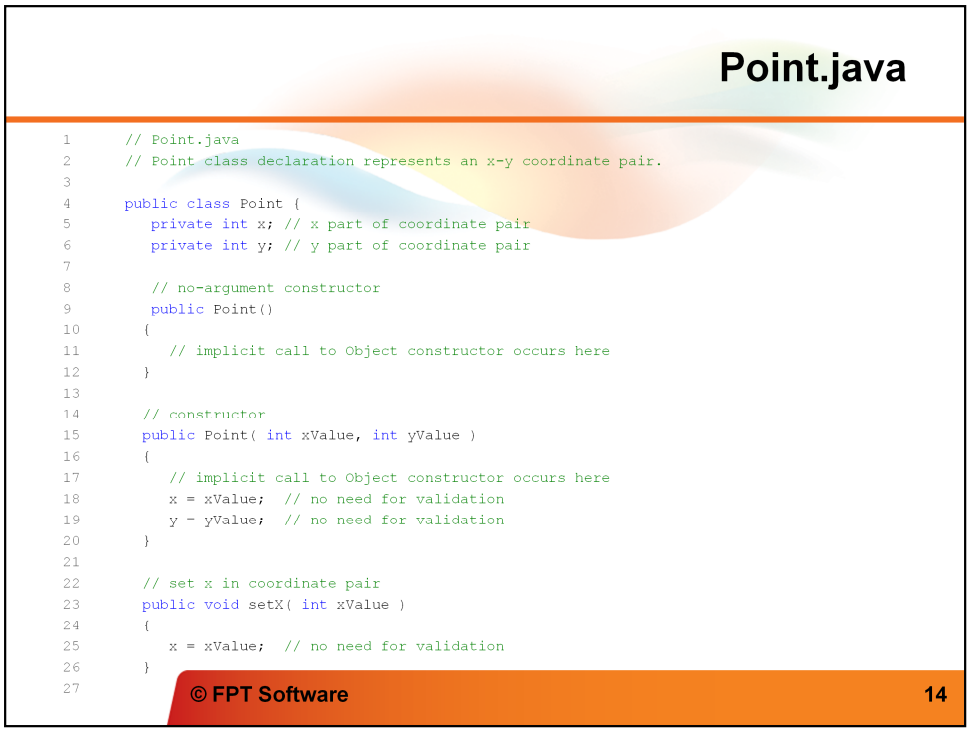
Inheritance hierarchy for Shapes.

protected Members

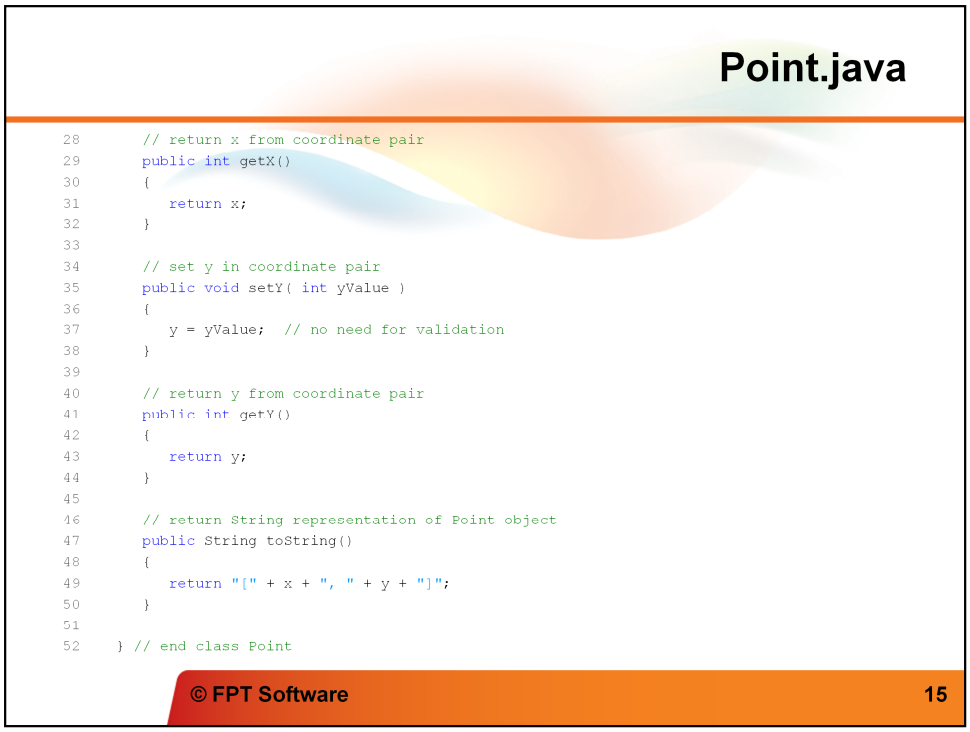
- protected access
  - Intermediate level of protection between public and private
  - protected members accessible to
    - superclass members
    - subclass members
    - Class members in the same package
  - Subclass access superclass member
    - Keyword super and a dot (.)
    - There is no super.super….

Superclasses - Subclasses relationship

- Superclass and subclass relationship
  - Example: Point/circle inheritance hierarchy
    - Point
      - x-y coordinate pair
    - Circle
      - x-y coordinate pair
      - Radius



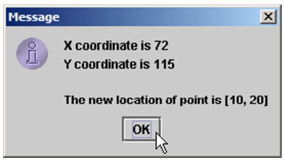
Lines 5-6: Maintain x- and y-coordinates as private instance variables.
Line 11: Implicit call to Object constructor



Lines 47-50: Override method `toString()` of class `Object`.

PointTest.java

```
1   // PointTest.java
2   // Testing class Point.
3   import javax.swing.JOptionPane;
4
5   public class PointTest {
6
7      public static void main( String[] args )
8      {
9         Point point = new Point( 72, 115 );  // create Point object
10
11       // get point coordinates
12       String output = "X coordinate is " + point.getX() +
13          "\nY coordinate is " + point.getY();
14
15       point.setX( 10 );  // set x-coordinate
16       point.setY( 20 );  // set y-coordinate
17
18       // get String representation of new point value
19       output += "\n\nThe new location of point is " + point;
20
21       JOptionPane.showMessageDialog( null, output ); // display output
22
23       System.exit( 0 );
24
25    } // end main
26
27 } // end class PointTest
```

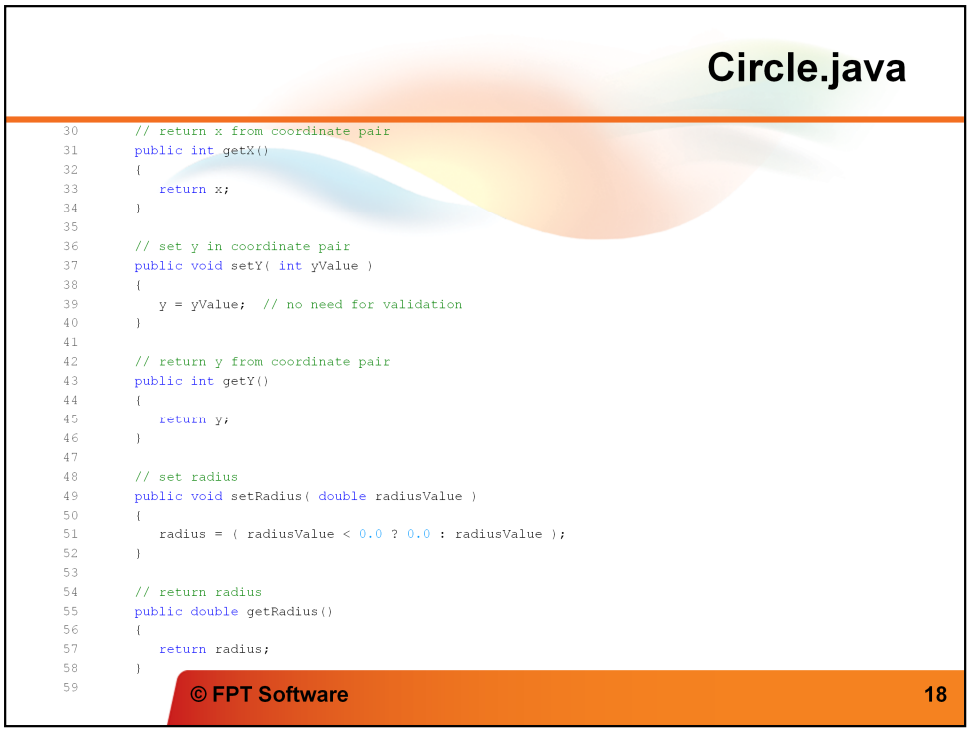


Line 9: Instantiate `Point` object
Lines 15-16 : Change the value of point's x- and y- coordinates
Line 19: Implicitly call point's `toString()` method

Circle.java

```
1   // Circle.java
2   // Circle class contains x-y coordinate pair and radius.
3
4   public class Circle {
5      private int x;          // x-coordinate of Circle's center
6      private int y;          // y-coordinate of Circle's center
7      private double radius;  // Circle's radius
8
9      // no-argument constructor
10     public Circle()
11     {
12        // implicit call to Object constructor occurs here
13     }
14
15     // constructor
16     public Circle( int xValue, int yValue, double radiusValue )
17     {
18        // implicit call to Object constructor occurs here
19        x = xValue;  // no need for validation
20        y = yValue;  // no need for validation
21        setRadius( radiusValue );
22     }
23
24     // set x in coordinate pair
25     public void setX( int xValue )
26     {
27        x = xValue;  // no need for validation
28     }
29
```


Lines 5-7: Maintain x- and y- coordinates and radius as `private` instance variables.
Lines 25-28: Note code similar to `Point` code.



Lines 31-47: Note code similar to `Point` code.
Line 51: Ensure non-negative value for `radius`

Circle.java

```
60      // calculate and return diameter
61      public double getDiameter()
62      {
63         return 2 * radius;
64      }
65
66      // calculate and return circumference
67      public double getCircumference()
68      {
69         return Math.PI * getDiameter();
70      }
71
72      // calculate and return area
73      public double getArea()
74      {
75         return Math.PI * radius * radius;
76      }
77
78      // return String representation of Circle object
79      public String toString()
80      {
81         return "Center = [" + x + ", " + y + "]; Radius = " + radius;
82      }
83
84   } // end class Circle
```

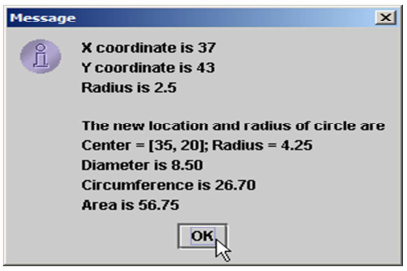
CircleTest.java

```
1  // CircleTest.java
2  // Testing class Circle.
3  import java.text.DecimalFormat;
4  import javax.swing.JOptionPane;
5
6  public class CircleTest {
7
8     public static void main( String[] args )
9     {
10       Circle circle = new Circle( 37, 43, 2.5 ); // create Circle object
11
12       // get Circle's initial x-y coordinates and radius
13       String output = "X coordinate is " + circle.getX() +
14          "\nY coordinate is " + circle.getY() +
15          "\nRadius is " + circle.getRadius();
16
17       circle.setX( 35 );         // set new x-coordinate
18       circle.setY( 20 );         // set new y-coordinate
19       circle.setRadius( 4.25 );  // set new radius
20
21       // get String representation of new circle value
22       output += "\n\nThe new location and radius of circle are\n" +
23          circle.toString();
24
25       // format floating-point values with 2 digits of precision
26       DecimalFormat twoDigits = new DecimalFormat( "0.00" );
27
```

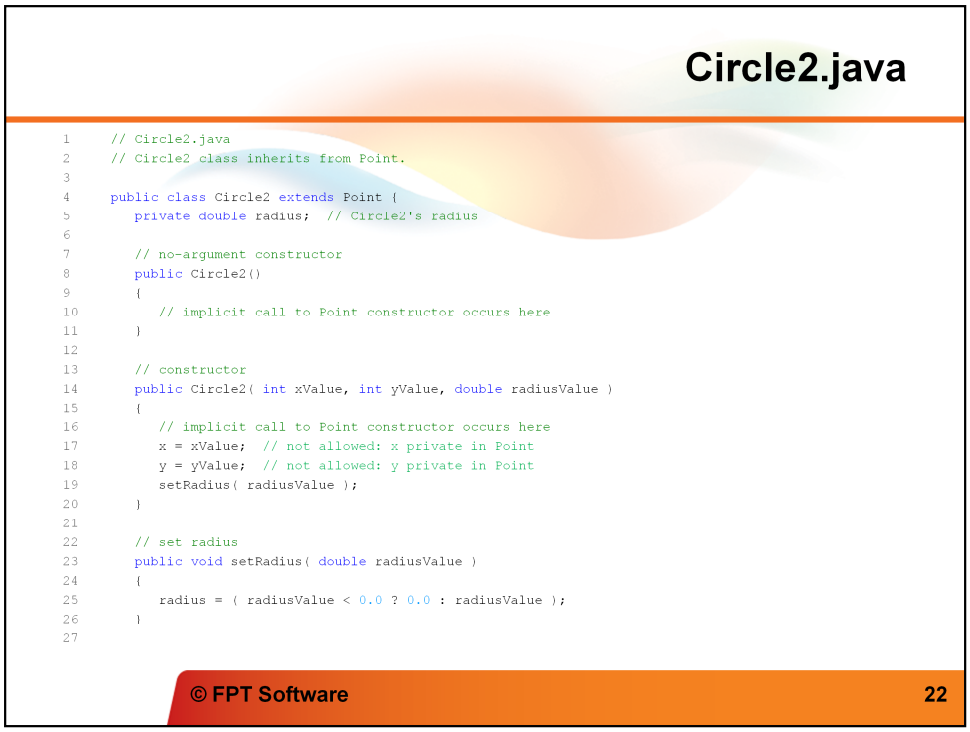

Line 10: Create `Circle` object
Lines 17-19: Use set methods to modify `private` instance variable
Line 23: Explicitly call circle's `toString` method

CircleTest.java

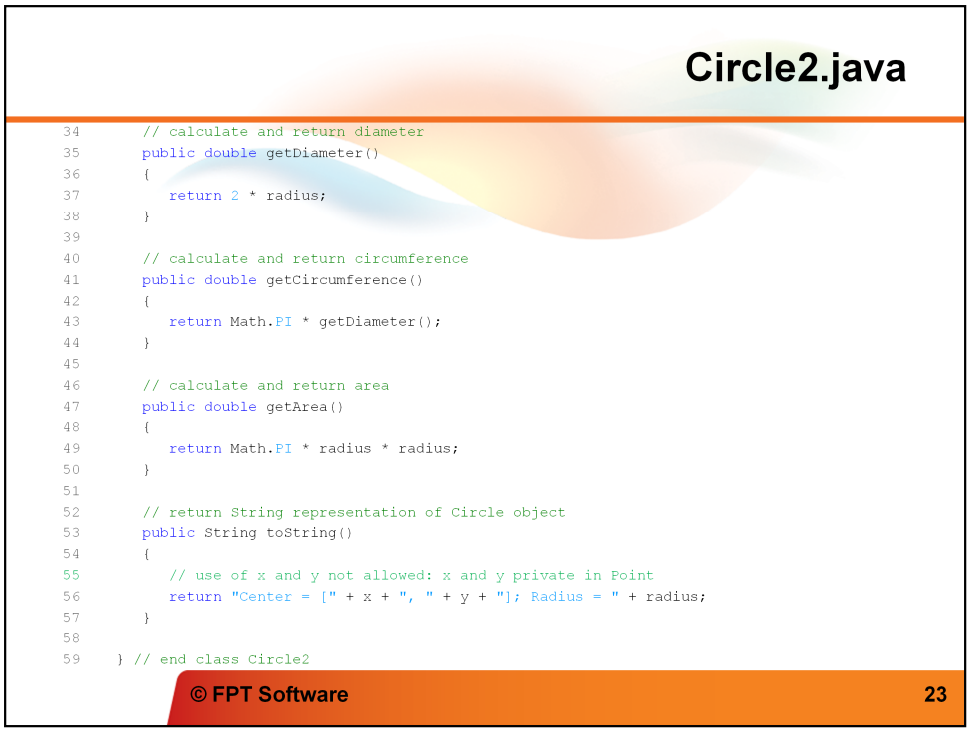
```
28          // get Circle's diameter
29          output += "\nDiameter is " +
30             twoDigits.format( circle.getDiameter() );
31
32          // get Circle's circumference
33          output += "\nCircumference is " +
34             twoDigits.format( circle.getCircumference() );
35
36          // get Circle's area
37          output += "\nArea is " + twoDigits.format( circle.getArea() );
38
39          JOptionPane.showMessageDialog( null, output ); // display output
40
41          System.exit( 0 );
42
43       } // end main
44
45    } // end class CircleTest
```



Lines 29-37: Use get methods to obtain circle's diameter, circumference and area.



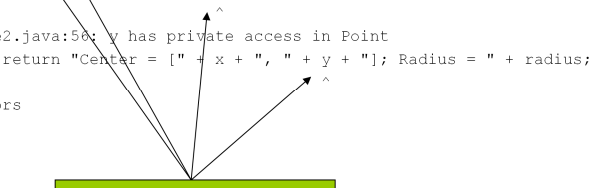
Line 4: Class `Circle2` extends class `Point`.
Line 5: Maintain `private` instance variable `radius`.
Lines 17-18: Attempting to access superclass `Point`'s `private` instance variables `x` and `y` results in syntax errors.



Line 56: Attempting to access superclass `Point`'s `private` instance variables `x` and `y` results in syntax errors.

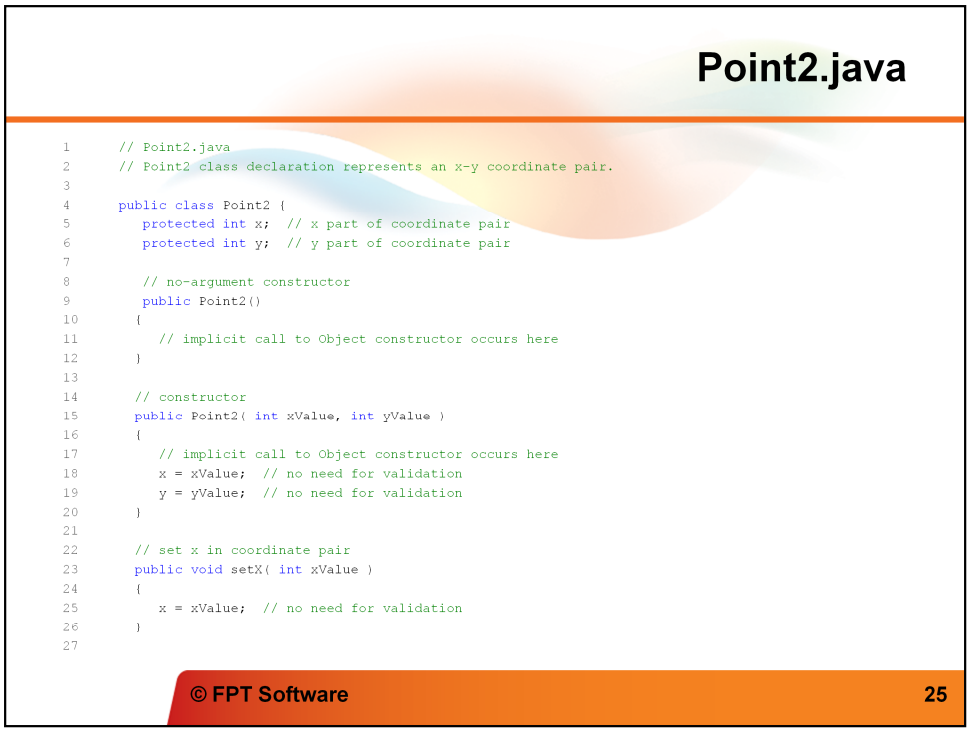
Circle2.java output

```
Circle2.java:17: x has private access in Point
      x = xValue;  // not allowed: x private in Point
      ^
Circle2.java:18: y has private access in Point
      y = yValue;  // not allowed: y private in Point
      ^
Circle2.java:56: x has private access in Point
      return "Center = [" + x + ", " + y + "]; Radius = " + radius;
                            ^
Circle2.java:56: y has private access in Point
      return "Center = [" + x + ", " + y + "]; Radius = " + radius;
                                   ^
4 errors
```



Attempting to access superclass `Point`'s `private` instance variables `x` and `y` results in syntax errors.

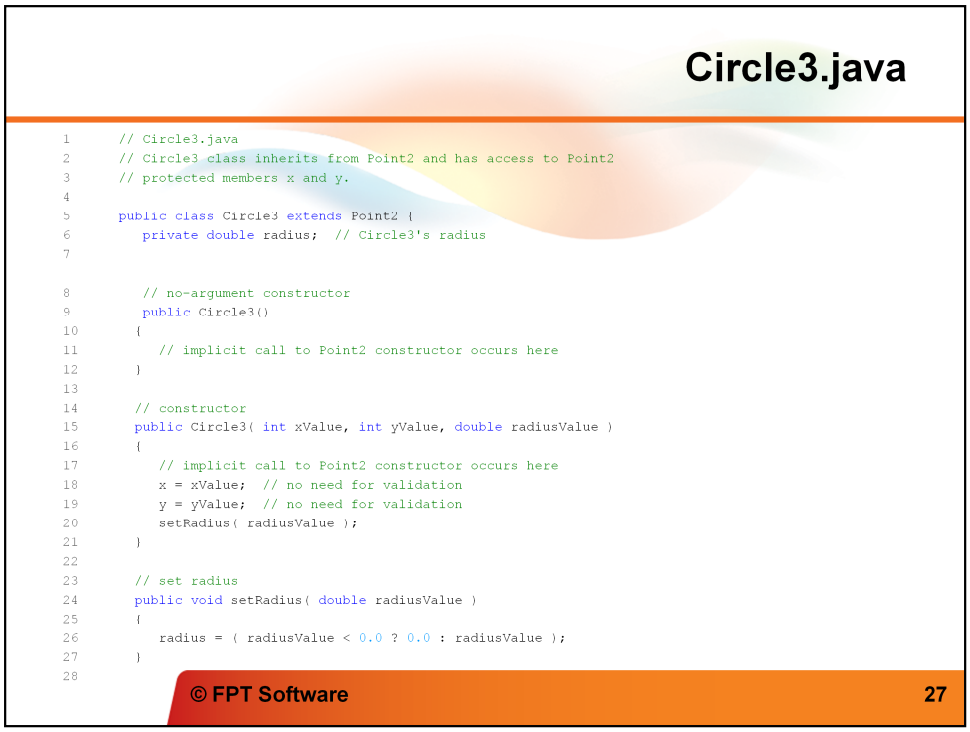
Attempting to access superclass `Point`'s `private` instance variables `x` and `y` results in syntax errors.



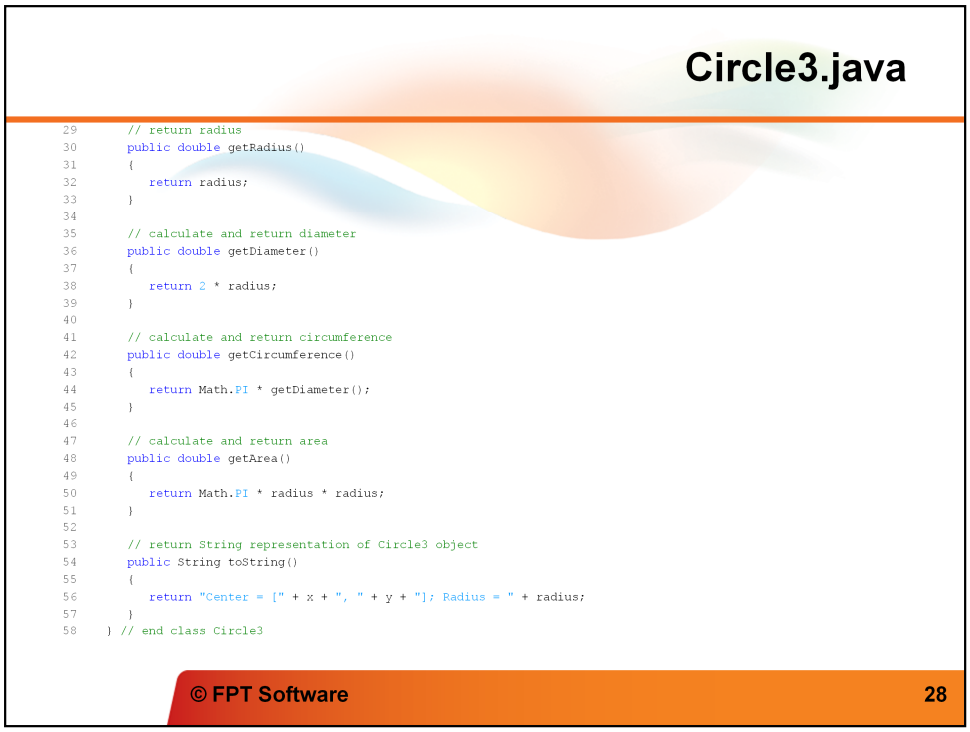
Lines 5-6: Maintain `x`- and `y`-coordinates as `protected` instance variables, accessible to subclasses.

Point2.java

```
28       // return x from coordinate pair
29       public int getX()
30       {
31          return x;
32       }
33
34       // set y in coordinate pair
35       public void setY( int yValue )
36       {
37          y = yValue;  // no need for validation
38       }
39
40       // return y from coordinate pair
41       public int getY()
42       {
43          return y;
44       }
45
46       // return String representation of Point2 object
47       public String toString()
48       {
49          return "[" + x + ", " + y + "]";
50       }
51
52    } // end class Point2
```



Line 5: Class `Circle3` inherits from class `Point2`.
Line 6: Maintain `private` instance variables `radius`.
Lines 11 and 17: Implicitly call superclass's default constructor.
Lines 18-19: Modify inherited instance variables `x` and `y`, declared `protected` in superclass `Point2`.



Line 56: Access inherited instance variables `x` and `y`, declared `protected` in superclass `Point2`.

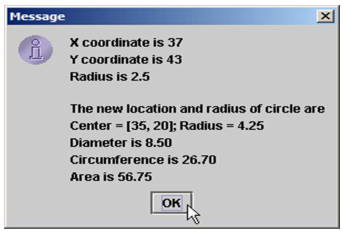
CircleTest3.java

```
1   // CircleTest3.java
2   // Testing class Circle3.
3   import java.text.DecimalFormat;
4   import javax.swing.JOptionPane;
5
6   public class CircleTest3 {
7
8      public static void main( String[] args )
9      {
10       // instantiate Circle object
11       Circle3 circle = new Circle3( 37, 43, 2.5 );
12
13       // get Circle3's initial x-y coordinates and radius
14       String output = "X coordinate is " + circle.getX() +
15          "\nY coordinate is " + circle.getY() +
16          "\nRadius is " + circle.getRadius();
17
18       circle.setX( 35 );         // set new x-coordinate
19       circle.setY( 20 );         // set new y-coordinate
20       circle.setRadius( 4.25 );  // set new radius
21
22       // get String representation of new circle value
23       output += "\n\nThe new location and radius of circle are\n" +
24          circle.toString();
25
```


Line 11: Create `Circle3` object.
Lines 14-15: Use inherited get methods to access inherited `protected` instance variables `x` and `y`.
Line 16: Use `Circle3` get method to access `private` instance variables.
Lines 18-19: Use inherited set methods to modify inherited protected data x and y.
Line 20: Use `Circle3` set method to modify `private` data `radius`.

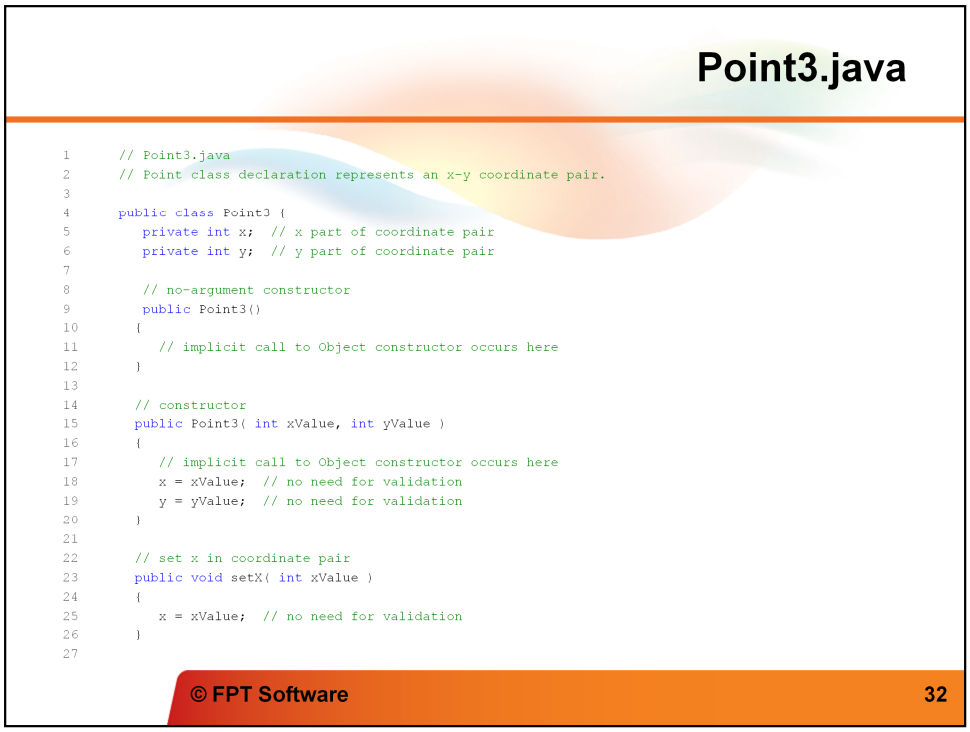
CircleTest3.java

```
26          // format floating-point values with 2 digits of precision
27          DecimalFormat twoDigits = new DecimalFormat( "0.00" );
28
29          // get Circle's diameter
30          output += "\nDiameter is " +
31             twoDigits.format( circle.getDiameter() );
32
33          // get Circle's circumference
34          output += "\nCircumference is " +
35             twoDigits.format( circle.getCircumference() );
36
37          // get Circle's area
38          output += "\nArea is " + twoDigits.format( circle.getArea() );
39
40          JOptionPane.showMessageDialog( null, output ); // display output
41
42          System.exit( 0 );
43
44       } // end method main
45
46    } // end class CircleTest3
```

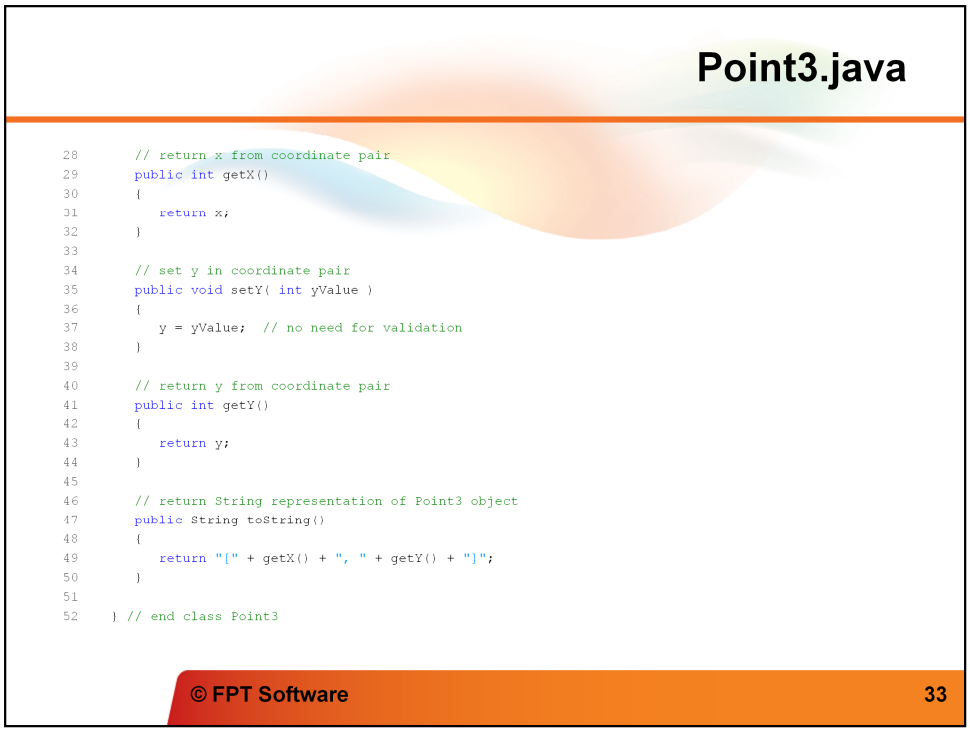


Relationship between Superclasses and Subclasses (Cont.)

- Using `protected` instance variables
  - Advantages
    - subclasses can modify values directly
    - Slight increase in performance
      - Avoid set/get function call overhead
  - Disadvantages
    - No validity checking
      - subclass can assign illegal value
    - Implementation dependent
      - subclass methods more likely dependent on superclass implementation
      - superclass implementation changes may result in subclass modifications
        - » Fragile (brittle) software



Lines 5-6: Better software-engineering practice: `private` over `protected` when possible.

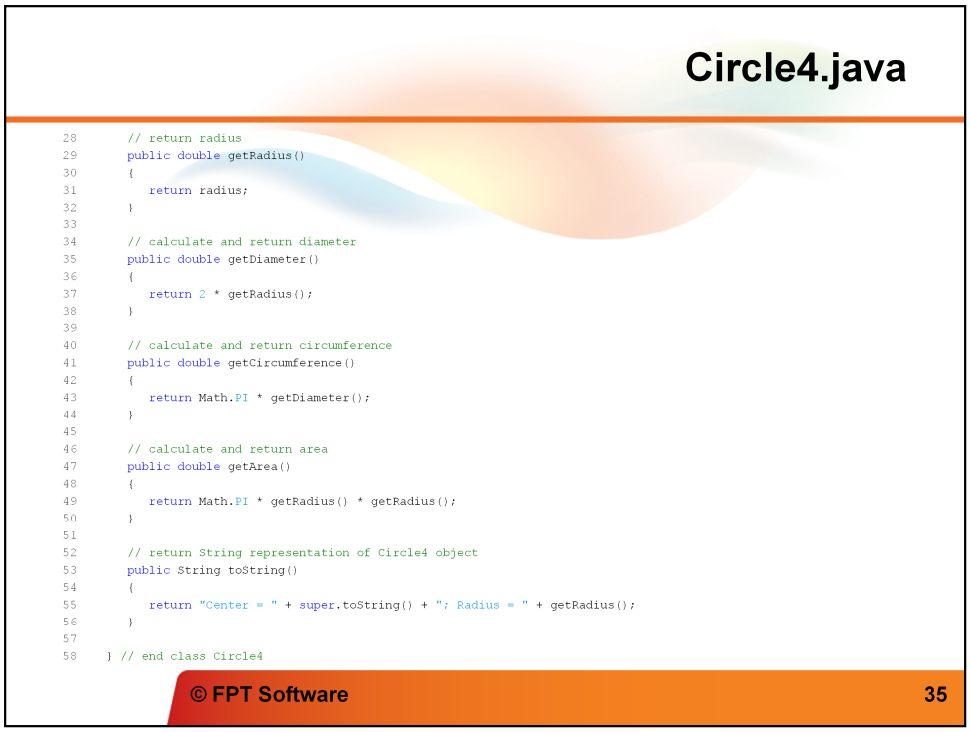


Line 49: Invoke `public` methods to access `private` instance variables.

Circle4.java

```
1   // Circle4.java
2   // Circle4 class inherits from Point3 and accesses Point3's
3   // private x and y via Point3's public methods.
4
5   public class Circle4 extends Point3 {
6
7      private double radius;  // Circle4's radius
8
9      // no-arqument constructor
10     public Circle4()
11     {
12        // implicit call to Point3 constructor occurs here
13     }
14
15     // constructor
16     public Circle4( int xValue, int yValue, double radiusValue )
17     {
18        super( xValue, yValue );  // call Point3 constructor explicitly
19        setRadius( radiusValue );
20     }
21
22     // set radius
23     public void setRadius( double radiusValue )
24     {
25        radius = ( radiusValue < 0.0 ? 0.0 : radiusValue );
26     }
27
```


Line 5: Class `Circle4` inherits from class `Point3`.
Line 7: Maintain `private` instance variable `radius`.



Line 37, 49 and 55: Invoke method `getRadius` rather than directly accessing instance variable `radius`.
Lines 53-56: Redefine class `Point3`'s method toString.

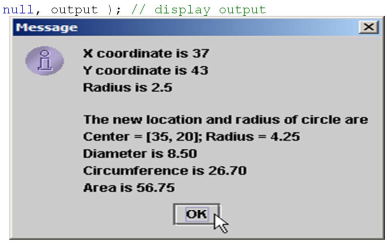
Circletest4.java

```
1   // CircleTest4.java
2   // Testing class Circle4.
3   import java.text.DecimalFormat;
4   import javax.swing.JOptionPane;
5
6   public class CircleTest4 {
7
8      public static void main( String[] args )
9      {
10       // instantiate Circle object
11       Circle4 circle = new Circle4( 37, 43, 2.5 );
12
13       // get Circle4's initial x-y coordinates and radius
14       String output = "X coordinate is " + circle.getX() +
15          "\nY coordinate is " + circle.getY() +
16          "\nRadius is " + circle.getRadius();
17
18       circle.setX( 35 );         // set new x-coordinate
19       circle.setY( 20 );         // set new y-coordinate
20       circle.setRadius( 4.25 );  // set new radius
21
22       // get String representation of new circle value
23       output += "\n\nThe new location and radius of circle are\n" +
24          circle.toString();
25
```


Line 11: Create `Circle4` object.
Lines 14 and 15: Use inherited get methods to access inherited `private` instance variables x and y.
Line 16: Use `Circle4` get method to access `private` instance variable radius.
Lines 18-19: Use inherited seta methods to modify inherited `private` instance variables x and y.
Line 20: Use `Circle4` set method to modify `private` instance variable radius.

Circletest4.java

```
26          // format floating-point values with 2 digits of precision
27          DecimalFormat twoDigits = new DecimalFormat( "0.00" );
28
29          // get Circle's diameter
30          output += "\nDiameter is " +
31             twoDigits.format( circle.getDiameter() );
32
33          // get Circle's circumference
34          output += "\nCircumference is " +
35             twoDigits.format( circle.getCircumference() );
36
37          // get Circle's area
38          output += "\nArea is " + twoDigits.format( circle.getArea() );
39
40          JOptionPane.showMessageDialog( null, output ); // display output
41
42          System.exit( 0 );
43
44       } // end main
45
46    } // end class CircleTest4
```

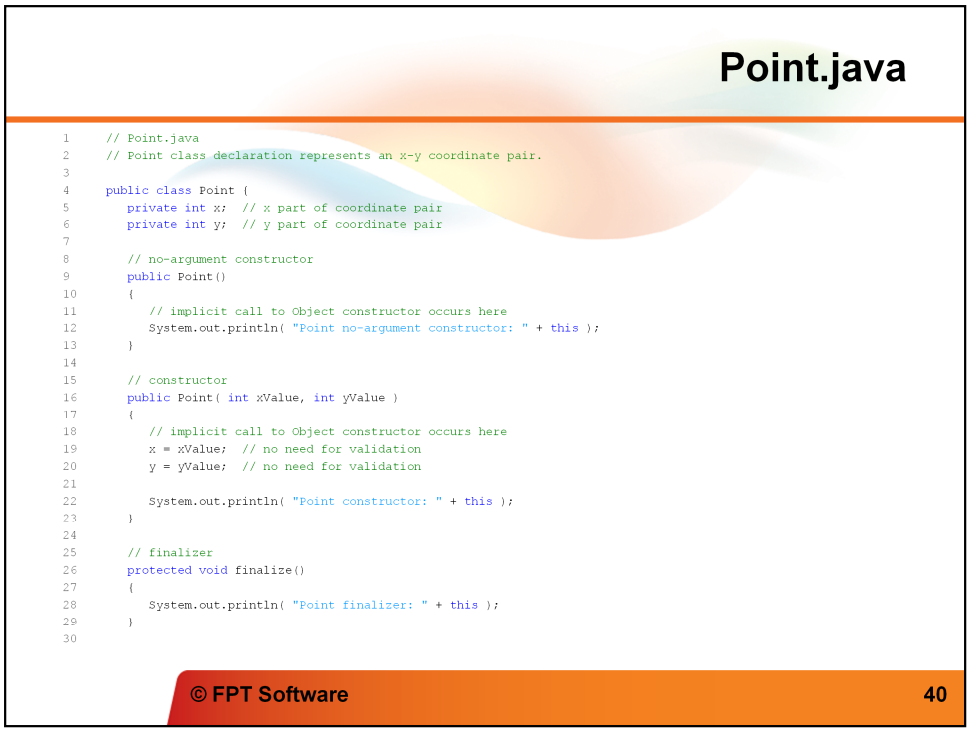


Constructors and Finalizers in Subclasses

- **Instantiating subclass object**
  - Chain of constructor calls
    - subclass constructor invokes superclass constructor
      - Implicitly or explicitly
    - Base of inheritance hierarchy
      - Last constructor called in chain is Object's constructor
      - Original subclass constructor's body finishes executing last
      - Example: Point3/Circle4/Cylinder hierarchy
        - » Point3 constructor called second last (last is Object constructor)
        - » Point3 constructor's body finishes execution second (first is Object constructor's body)

Constructors and Destructors in Derived Classes

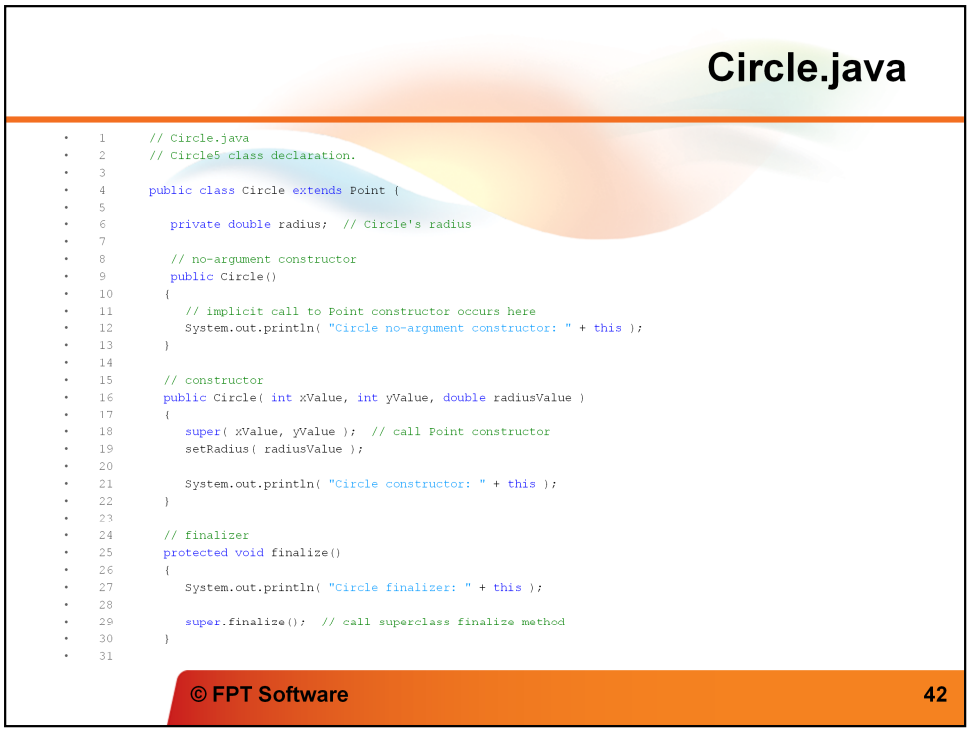
- Garbage collecting subclass object
  - Chain of `finalize` method calls
    - Reverse order of constructor chain
    - Finalizer of subclass called first
    - Finalizer of next superclass up hierarchy next
      - Continue up hierarchy until final superreached
        - » After final superclass (`Object`) finalizer, object removed from memory



Lines 12, 22 and 28: Constructor and finalizer output messages to demonstrate method call order.

Point.java

```
31      // set x in coordinate pair
32      public void setX( int xValue )
33      {
34         x = xValue;  // no need for validation
35      }
36
37      // return x from coordinate pair
38      public int getX()
39      {
40         return x;
41      }
42
43      // set y in coordinate pair
44      public void setY( int yValue )
45      {
46         y = yValue;  // no need for validation
47      }
48
49      // return y from coordinate pair
50      public int getY()
51      {
52         return y;
53      }
54
55      // return String representation of Point4 object
56      public String toString()
57      {
58         return "[" + getX() + ", " + getY() + "]";
59      }
60
61   } // end class Point
```



Lines 12, 21 and 29: Constructor and finalizer output messages to demonstrate method call order.

Circle.java

```
32      // set radius
33      public void setRadius( double radiusValue )
34      {
35         radius = ( radiusValue < 0.0 ? 0.0 : radiusValue );
36      }
37
38      // return radius
39      public double getRadius()
40      {
41         return radius;
42      }
43
44      // calculate and return diameter
45      public double getDiameter()
46      {
47         return 2 * getRadius();
48      }
49
50      // calculate and return circumference
51      public double getCircumference()
52      {
53         return Math.PI * getDiameter();
54      }
```

Circle.java

```
55
56       // calculate and return area
57       public double getArea()
58       {
59          return Math.PI * getRadius() * getRadius();
60       }
61
62       // return String representation of Circle5 object
63       public String toString()
64       {
65          return "Center = " + super.toString() + "; Radius = " + getRadius();
66       }
67
68    } // end class Circle
```

ConstructorFinalizerTest.java

```
1   // ConstructorFinalizerTest.java
2   // Display order in which superclass and subclass
3   // constructors and finalizers are called.
4
5   public class ConstructorFinalizerTest {
6
7      public static void main( String args[] )
8      {
9         Point point;
10        Circle circle1, circle2;
11
12        point = new Point( 11, 22 );
13
14        System.out.println();
15        circle1 = new Circle( 72, 29, 4.5 );
16
17        System.out.println();
18        circle2 = new Circle( 5, 7, 10.67 );
19
20        point = null;    // mark for garbage collection
21        circle1 = null;  // mark for garbage collection
22        circle2 = null;  // mark for garbage collection
23
24        System.out.println();
25
```


Line 12: `Point` object goes in and out of scope immediately.
Lines 15 and 18: Instantiate two `Circle` objects to demonstrate order of subclass and superclass constructor/finalizer method calls.

ConstructorFinalizerTest.java

```
26          System.gc();  // call the garbage collector
27
28       } // end main
29
30    } // end class ConstructorFinalizerTest
```

```
Point constructor: [11, 22]

Point constructor: Center = [72, 29]; Radius = 0.0
Circle constructor: Center = [72, 29]; Radius = 4.5

Point constructor: Center = [5, 7]; Radius = 0.0
Circle constructor: Center = [5, 7]; Radius = 10.67

Point finalizer: [11, 22]
Circle finalizer: Center = [72, 29]; Radius = 4.5
Point finalizer: Center = [72, 29]; Radius = 4.5
Circle finalizer: Center = [5, 7]; Radius = 10.67
Point finalizer: Center = [5, 7]; Radius = 10.67
```

Subclass `Circle` constructor body executes after superclass `Point4`'s constructor finishes execution.

Finalizer for `Circle` object called in reverse order of constructors.

Summary

- Inheritance is a mechanism that allows one class to reuse the implementation provided by another.
- A class always **extends** exactly one superclass. If a class does not explicitly extend another, it implicitly extends the class Object.
- A superclass method or field can be accessed using a **super**. keyword.
- Subclass objects can not access superclass's private data unless they change into **protected** access level.
- If a constructor does not explicitly invoke another (this() or super()) constructor, it implicitly invokes the superclass's no-args constructor

Session 2

POLYMORPHISM

Polymorphism

- Polymorphism is a generic term for having many forms. You can use the same name for several different things and the compiler automatically figures out which version you wanted. There are several forms of polymorphism supported in Java, shadowing, overriding, and overloading
- What does it mean:
  - "Program in the general"
  - Treat objects in same class hierarchy as if all superclass
  - Makes programs extensible
    - New classes added easily, can still be processed

Relationships Among Objects in an Inheritance Hierarchy

- **Previously**
  - Circle inherited from Point
  - Manipulated Point and Circle objects using references to invoke methods
- **This section**
  - Invoking superclass methods from subclass objects
  - Using superclass references with subclass-type variables
  - Subclass method calls via superclass-type variables
- **Key concept**
  - subclass object can be treated as superclass object
    - "is-a" relationship
    - superclass is not a subclass object

Invoking Superclass Methods from Subclass Objects

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

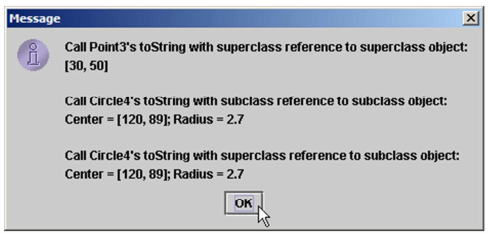
HierarchyRelationshipTest1.java

```
1   // HierarchyRelationshipTest1.java
2   // Assigning superclass and subclass references to superclass- and
3   // subclass-type variables.
4   import javax.swing.JOptionPane;
5
6   public class HierarchyRelationshipTest1 {
7
8      public static void main( String[] args )
9      {
10       // assign superclass reference to superclass-type variable
11       Point3 point = new Point3( 30, 50 );
12
13       // assign subclass reference to subclass-type variable
14       Circle4 circle = new Circle4( 120, 89, 2.7 );
15
16       // invoke toString on superclass object using superclass variable
17       String output = "Call Point3's toString with superclass" +
18          " reference to superclass object: \n" + point.toString();
19
20       // invoke toString on subclass object using subclass variable
21       output += "\n\nCall Circle4's toString with subclass" +
22          " reference to subclass object: \n" + circle.toString();
23
```


Line 11: Assign superclass reference to superclass-type variable
Line 14: Assign subclass reference to subclass-type variable
Line 17: Invoke `toString` on superclass object using superclass variable
Line 22: Invoke `toString` on subclass object using subclass variable

HierarchyRelationshipTest1.java

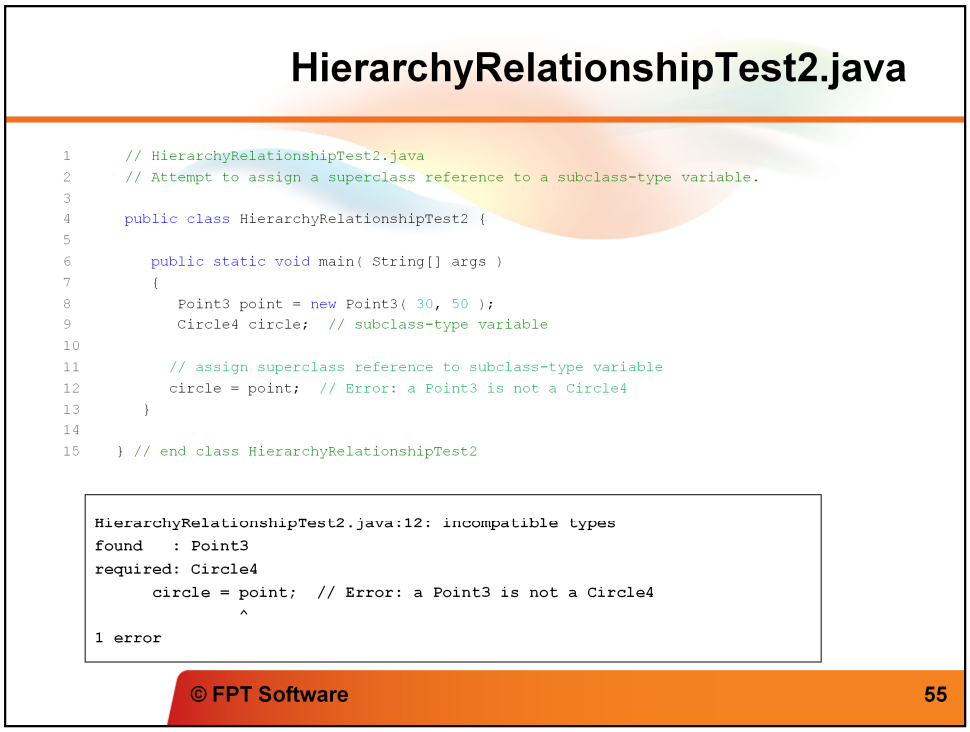
```
24          // invoke toString on subclass object using superclass variable
25          Point3 pointRef = circle;
26          output += "\n\nCall Circle4's toString with superclass" +
27             " reference to subclass object: \n" + pointRef.toString();
28
29          JOptionPane.showMessageDialog( null, output );  // display output
30
31          System.exit( 0 );
32
33       } // end main
34
35    } // end class HierarchyRelationshipTest1
```



Line 25: Assign subclass reference to superclass-type variable.
Line 27: Invoke `toString()` on subclass object using superclass variable.

Using Superclass References with Subclass-Type Variables

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



Line 12: Assigning superclass reference to subclass-type variable causes compiler error.

HierarchyRelationshipTest21.java

```
1     // HierarchyRelationshipTest21.java
2     // Attempt to downcasting a superclass to a subclass-type variable.
3
4     public class HierarchyRelationshipTest2 {
5
6        public static void main( String[] args )
7        {
8           Point3 point = new Point3( 30, 50 );
9           Circle4 circle;  // subclass-type variable
10
11          // assign superclass reference to subclass-type variable
12          circle = (Circle4) point;  // Try to downcasting a Point3 to Circle4
13       }
14
15    } // end class HierarchyRelationshipTest21
```

```
Exception in thread "main" java.lang.ClassCastException: Point3 cannot be
cast to Circle4
```


Line 12: `Exception in thread "main" java.lang.ClassCastException: Point3 cannot be cast to Circle4`

HierarchyRelationshipTest22.java

```
1   // HierarchyRelationshipTest22.java
2   // Attempt to downcasting a superclass to a subclass-type variable.
3
4   public class HierarchyRelationshipTest2 {
5
6      public static void main( String[] args )
7      {
8         Point3 point = new Circle4( 30, 50 , 8.5);
9         Circle4 circle;  // subclass-type variable
10
11        // assign superclass reference to subclass-type variable
12        circle = (Circle4) point;  // Casting a Point3 to Circle4 success
13     }
14
15  } // end class HierarchyRelationshipTest22
```

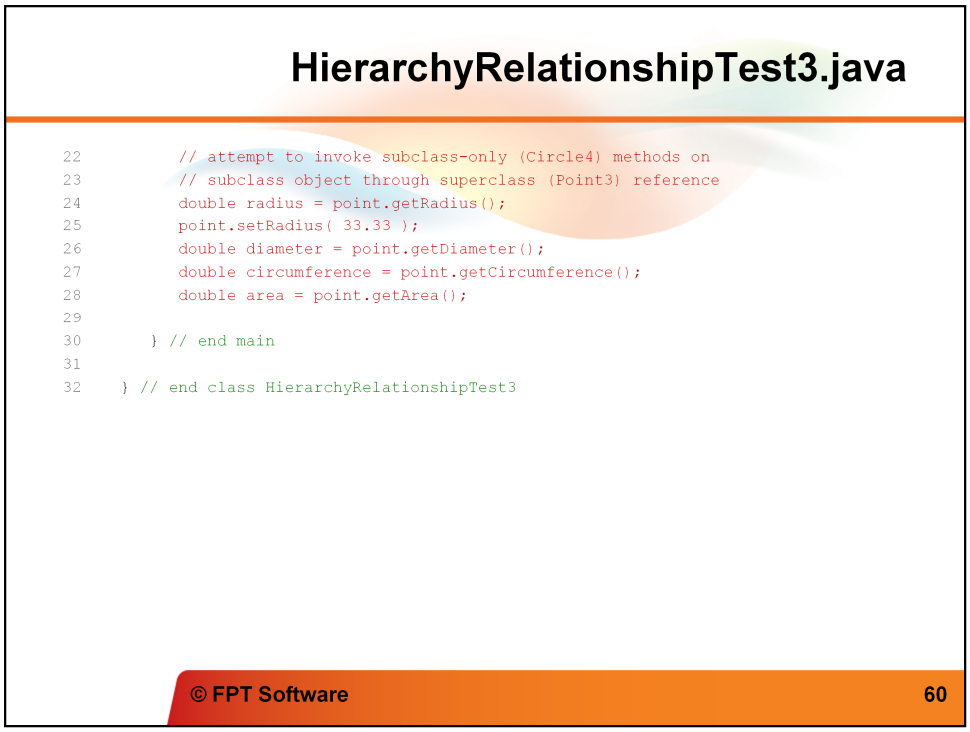

Line 8: variable “point” is a Circle4, so we can downcasting at line 12 without any error.

Subclass Method Calls via Superclass-Type variables

- Call a subclass method with superclass reference
  - Compiler error
    - Subclass methods are not superclass methods

HierarchyRelationshipTest3.java

```
1   // HierarchyRelationshipTest3.java
2   // Attempting to invoke subclass-only member methods through
3   // a superclass reference.
4
5   public class HierarchyRelationshipTest3 {
6
7      public static void main( String[] args )
8      {
9        Point3 point;
10       Circle4 circle = new Circle4( 120, 89, 2.7 );
11
12       point = circle;  // aim superclass reference at subclass object
13
14       // invoke superclass (Point3) methods on subclass
15       // (Circle4) object through superclass reference
16       int x = point.getX();
17       int y = point.getY();
18       point.setX( 10 );
19       point.setY( 20 );
20       point.toString();
21
```



Lines 24-28: Attempt to invoke subclass-only (`Circle4`) methods on subclass object through superclass (`Point3`) reference.

HierarchyRelationshipTest3.java

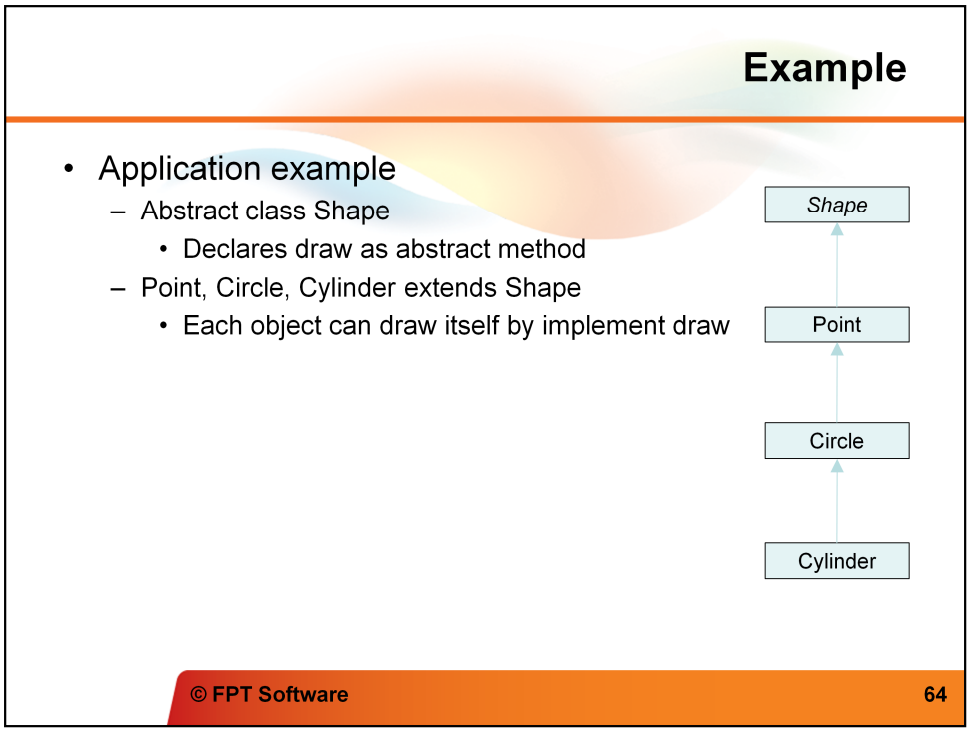
```
HierarchyRelationshipTest3.java:24: cannot resolve symbol
symbol  : method getRadius ()
location: class Point3
      double radius = point.getRadius();
                            ^
HierarchyRelationshipTest3.java:25: cannot resolve symbol
symbol  : method setRadius (double)
location: class Point3
      point.setRadius( 33.33 );
            ^
HierarchyRelationshipTest3.java:26: cannot resolve symbol
symbol  : method getDiameter ()
location: class Point3
      double diameter = point.getDiameter();
                              ^
HierarchyRelationshipTest3.java:27: cannot resolve symbol
symbol  : method getCircumference ()
location: class Point3
      double circumference = point.getCircumference();
                                   ^
HierarchyRelationshipTest3.java:28: cannot resolve symbol
symbol  : method getArea ()
location: class Point3
      double area = point.getArea();
                          ^
5 errors
```

Abstract Class

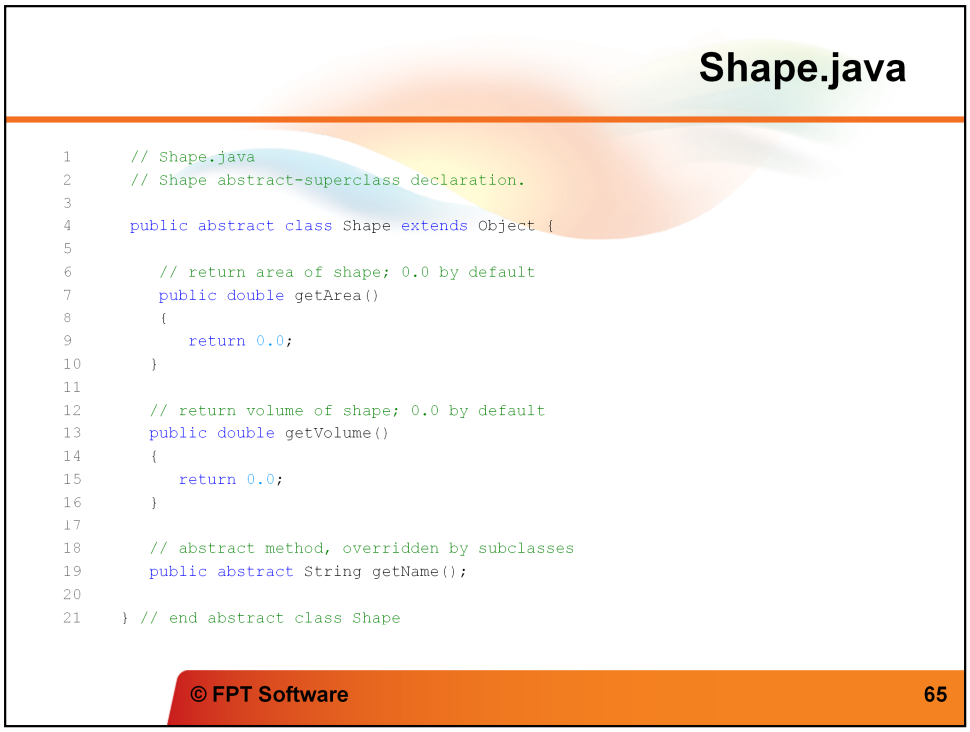
- Abstract class
  - Is declared `abstract`
  - Are superclasses (called abstract superclasses)
  - Cannot be instantiated, but they can be subclassed.
  - May or may not include abstract methods
  - When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, the subclass must also be declared abstract.

Abstract Method

- An *abstract method* is a method that is declared without an implementation (without braces, and followed by a semicolon).
  - Example: `abstract void moveTo(int x, int y);`
- If a class includes abstract methods, the class itself *must* be declared abstract.
- All of the methods in an *interface* are *implicitly* abstract, so the abstract modifier is not used with interface methods (it could be -it's just not necessary).



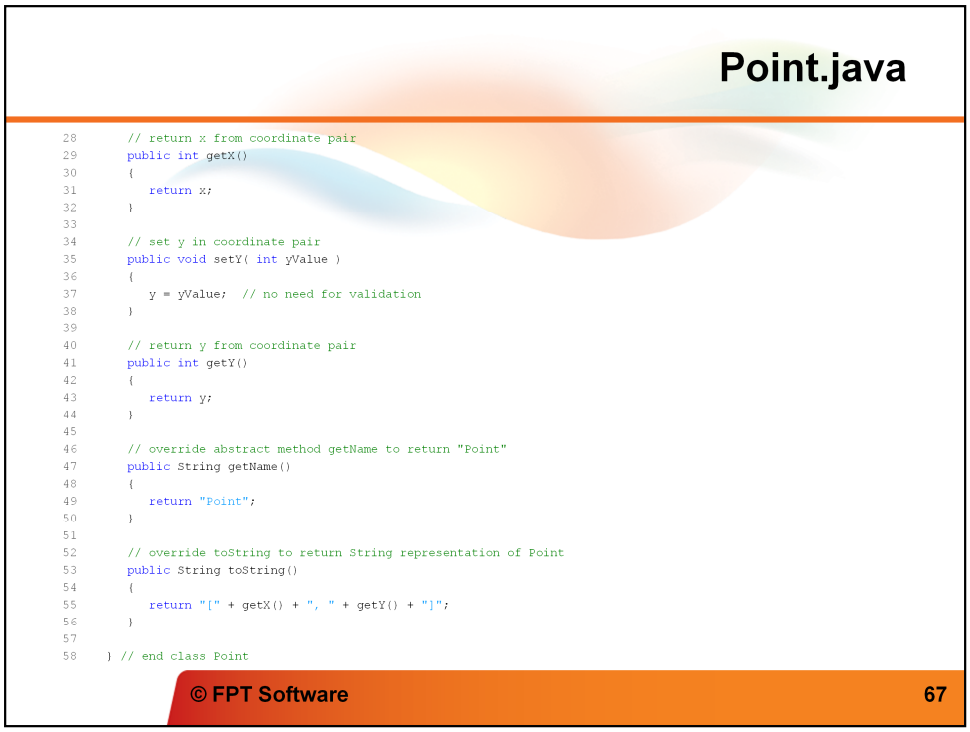
In the example we will use method getName() instead of draw() for simple.



Line 4: Keyword `abstract` declares class `Shape` as abstract class
Line 19: Keyword `abstract` declares method `getName` as abstract method

Point.java

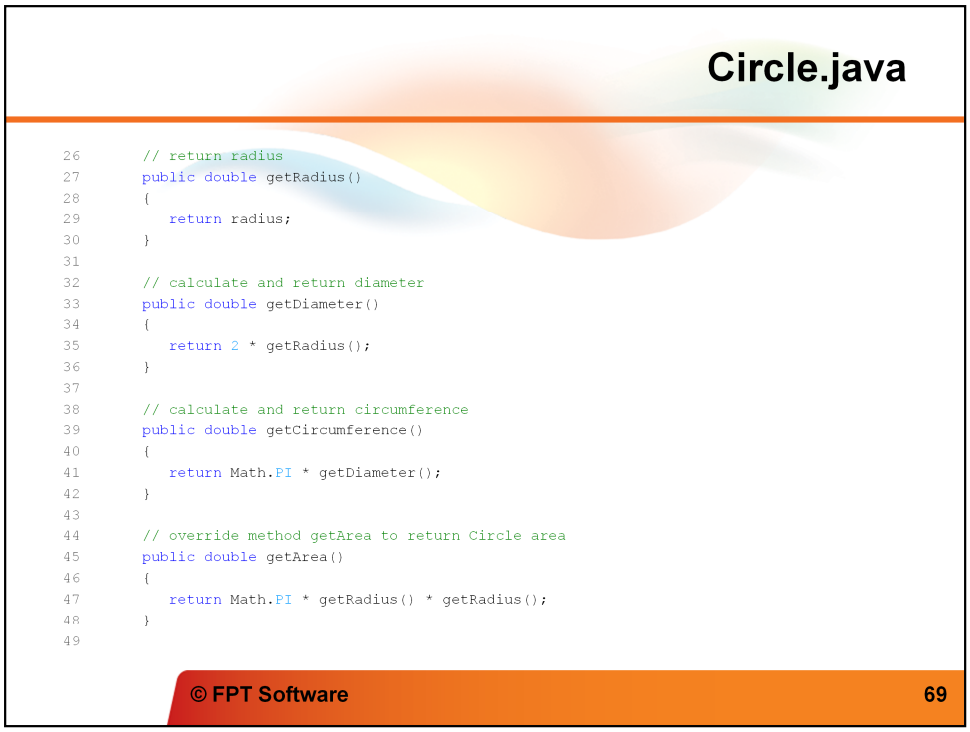
```
1   // Point.java
2   // Point class declaration inherits from Shape.
3
4   public class Point extends Shape {
5      private int x;  // x part of coordinate pair
6      private int y;  // y part of coordinate pair
7
8      // no-argument constructor; x and y default to 0
9      public Point()
10     {
11        // implicit call to Object constructor occurs here
12     }
13
14     // constructor
15     public Point( int xValue, int yValue )
16     {
17        // implicit call to Object constructor occurs here
18        x = xValue;  // no need for validation
19        y = yValue;  // no need for validation
20     }
21
22     // set x in coordinate pair
23     public void setX( int xValue )
24     {
25        x = xValue;  // no need for validation
26     }
27
```



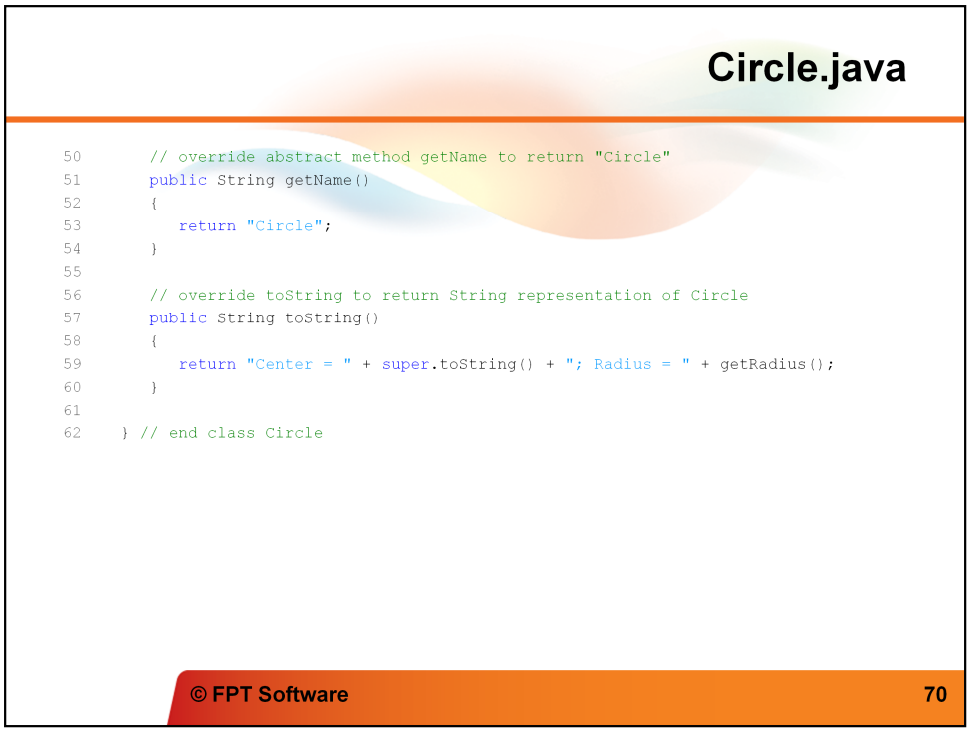
Lines 47-50: Override `abstract` method `getName`.

Circle.java

```
1   // Circle.java
2   // Circle class inherits from Point.
3
4   public class Circle extends Point {
5      private double radius;  // Circle's radius
6
7      // no-argument constructor; radius defaults to 0.0
8      public Circle()
9      {
10       // implicit call to Point constructor occurs here
11     }
12
13     // constructor
14     public Circle( int x, int y, double radiusValue )
15     {
16        super( x, y );  // call Point constructor
17        setRadius( radiusValue );
18     }
19
20     // set radius
21     public void setRadius( double radiusValue )
22     {
23        radius = ( radiusValue < 0.0 ? 0.0 : radiusValue );
24     }
25
```



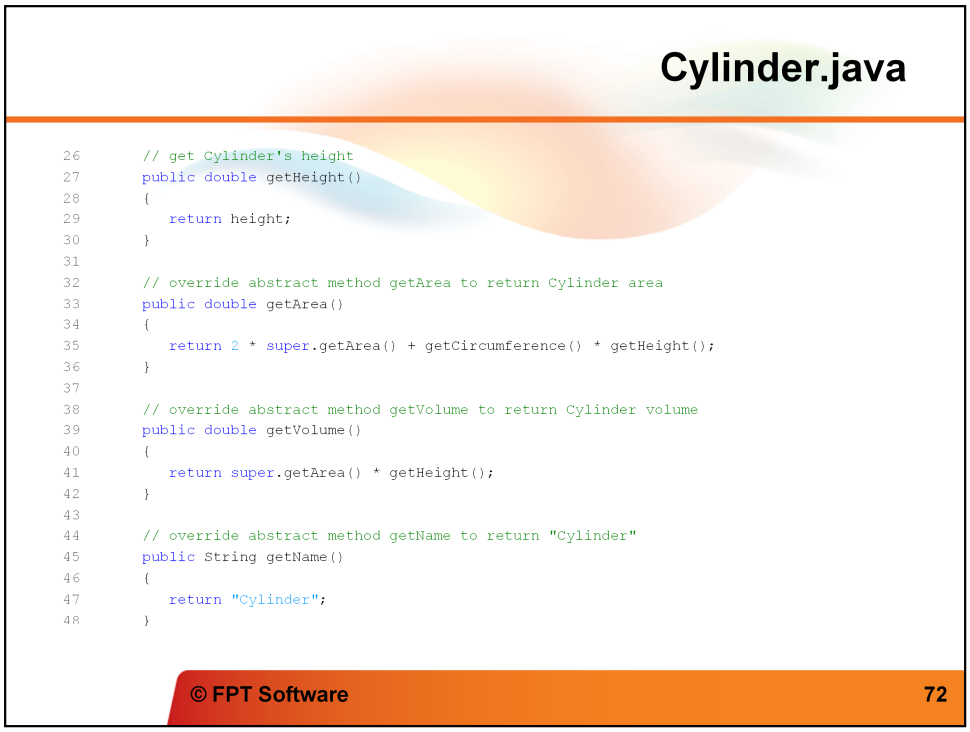
Lines 45-48: Override method `getArea` to return circle area.



Lines 51-54: Override `abstract` method `getName`.

Cylinder.java

```
1   // Cylinder.java
2   // Cylinder class inherits from Circle.
3
4   public class Cylinder extends Circle {
5      private double height;  // Cylinder's height
6
7      // no-argument constructor; height defaults to 0.0
8      public Cylinder()
9      {
10        // implicit call to Circle constructor occurs here
11     }
12
13     // constructor
14     public Cylinder( int x, int y, double radius, double heightValue )
15     {
16        super( x, y, radius );  // call Circle constructor
17        setHeight( heightValue );
18     }
19
20     // set Cylinder's height
21     public void setHeight( double heightValue )
22     {
23        height = ( heightValue < 0.0 ? 0.0 : heightValue );
24     }
25
```



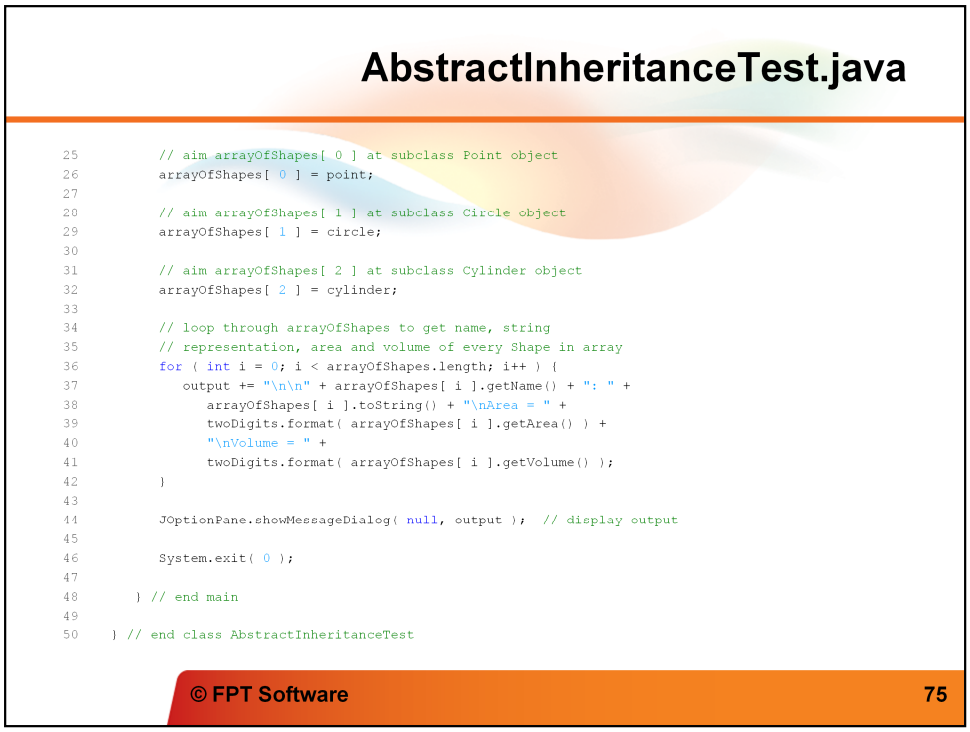
Lines 33-36: Override method `getArea` to return cylinder area
Lines 39-42: Override method `getVolume` to return cylinder volume
Lines 45-48: Override `abstract` method `getName`

Cylinder.java

```
49
50      // override toString to return String representation of Cylinder
51      public String toString()
52      {
53         return super.toString() + "; Height = " + getHeight();
54      }
55
56   } // end class Cylinder
```

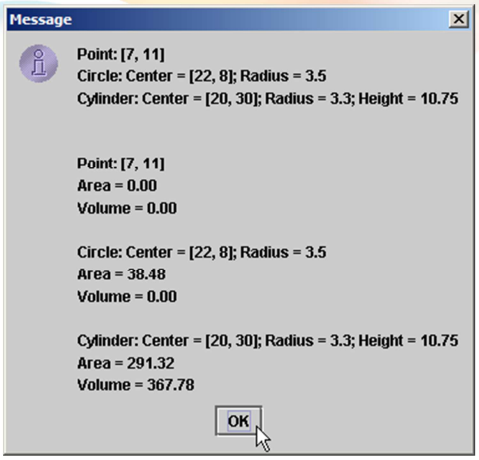
AbstractInheritanceTest.java

```
1   // AbstractInheritanceTest.java
2   // Driver for shape, point, circle, cylinder hierarchy.
3   import java.text.DecimalFormat;
4   import javax.swing.JOptionPane;
5
6   public class AbstractInheritanceTest {
7
8      public static void main( String args[] )
9      {
10       // set floating-point number format
11       DecimalFormat twoDigits = new DecimalFormat( "0.00" );
12
13       // create Point, Circle and Cylinder objects
14       Point point = new Point( 7, 11 );
15       Circle circle = new Circle( 22, 8, 3.5 );
16       Cylinder cylinder = new Cylinder( 20, 30, 3.3, 10.75 );
17
18       // obtain name and string representation of each object
19       String output = point.getName() + ": " + point + "\n" +
20          circle.getName() + ": " + circle + "\n" +
21          cylinder.getName() + ": " + cylinder + "\n";
22
23       Shape arrayOfShapes[] = new Shape[ 3 ];  // create Shape array
24
```



Lines 26-32: Create an array of generic `Shape` objects
Lines 36-42: Loop through `arrayOfShapes` to get name, string representation, area and volume of every shape in array

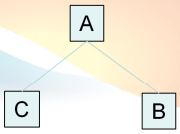
Output


Message
Point: [7, 11]
Circle: Center = [22, 8]; Radius = 3.5
Cylinder: Center = [20, 30]; Radius = 3.3; Height = 10.75
Point: [7, 11]
Area = 0.00
Volume = 0.00
Circle: Center = [22, 8]; Radius = 3.5
Area = 38.48
Volume = 0.00
Cylinder: Center = [20, 30]; Radius = 3.3; Height = 10.75
Area = 291.32
Volume = 367.78
OK

Interfaces

- Think of interface as a “pure” abstract class. It allows the creator to establish the form for a class: method names, argument lists and return types, but no method bodies.
- An **interface** says: “This is what all classes that *implement* this particular interface will look like.”

When to use interface



- Let B & C be classes. Assume we make A the parent class of B and C so A can hold the methods and fields that are common between B and C.
- We can make A an abstract classes. The methods in A then indicate which methods must be implemented in B and C. A can act as type, which can hold objects of type B or C
- Sometimes all the methods of B must be implemented differently than the same method in C. Make A an **interface.**
- An interface, like a class, defines a type

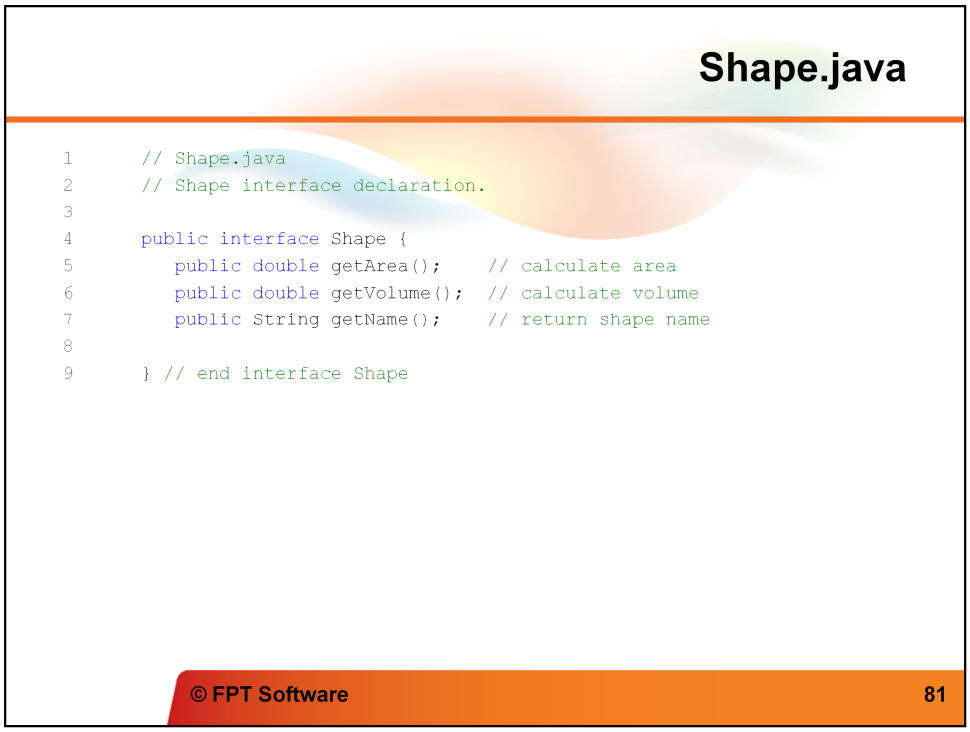
When to use interface

- Assume that C is a subclass of A and B. Since Java doesn't support multi inheritance, so A and B cannot be both abstract class. We must use interface here.

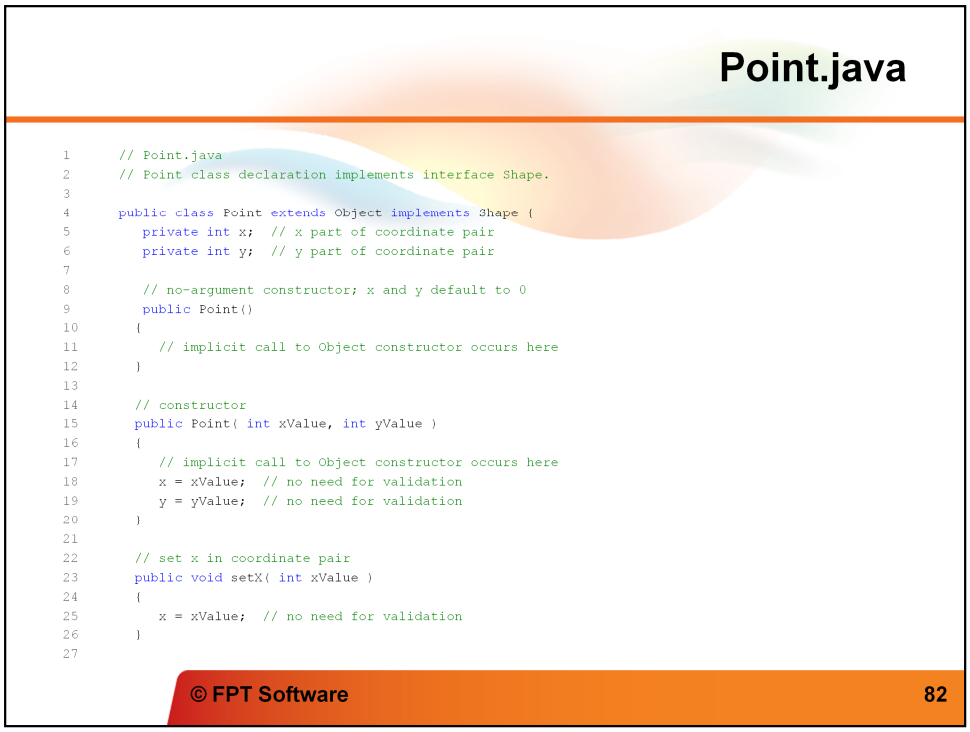
Note: diamond problem

Creating and Using Interfaces

- Use interface Shape
  - Replace abstract class Shape
- Interface
  - Declaration begins with interface keyword
  - Classes implement an interface (and its methods)
  - Contains public abstract methods
    - Classes (that implement the interface) must implement these methods
    - Abstract classes (that implement the interface) MAYNOT implement these methods



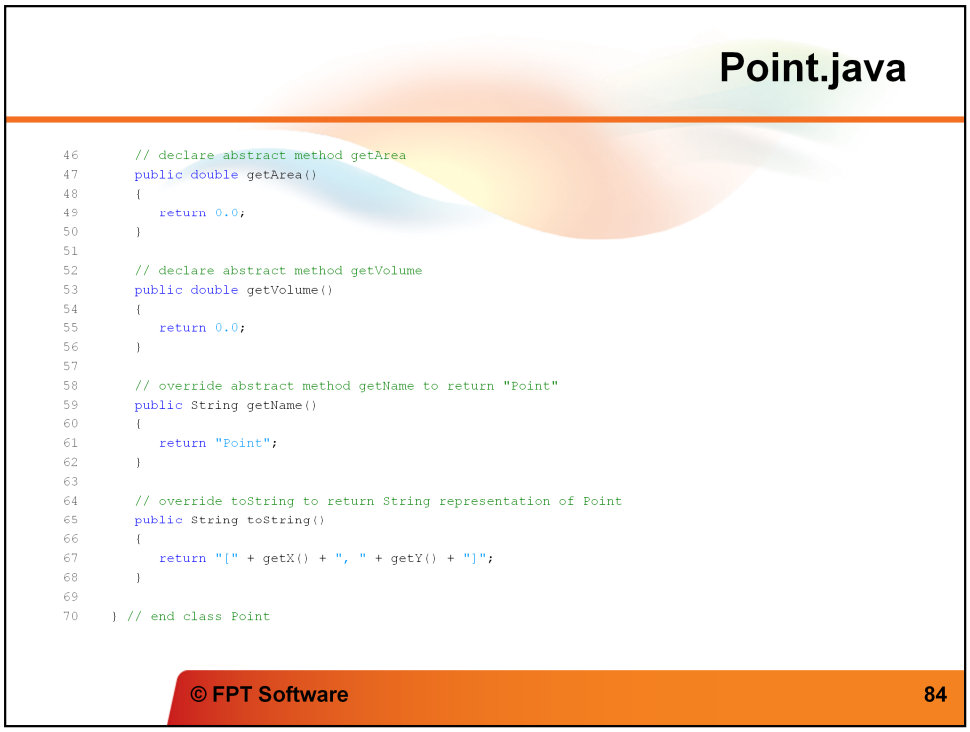
Lines 5-7: Classes that `implement Shape` must implement these methods



Line 4: `Point` implements interface `Shape`

Point.java

```
28       // return x from coordinate pair
29       public int getX()
30       {
31          return x;
32       }
33
34       // set y in coordinate pair
35       public void setY( int yValue )
36       {
37          y = yValue;  // no need for validation
38       }
39
40       // return y from coordinate pair
41       public int getY()
42       {
43          return y;
44       }
45
```

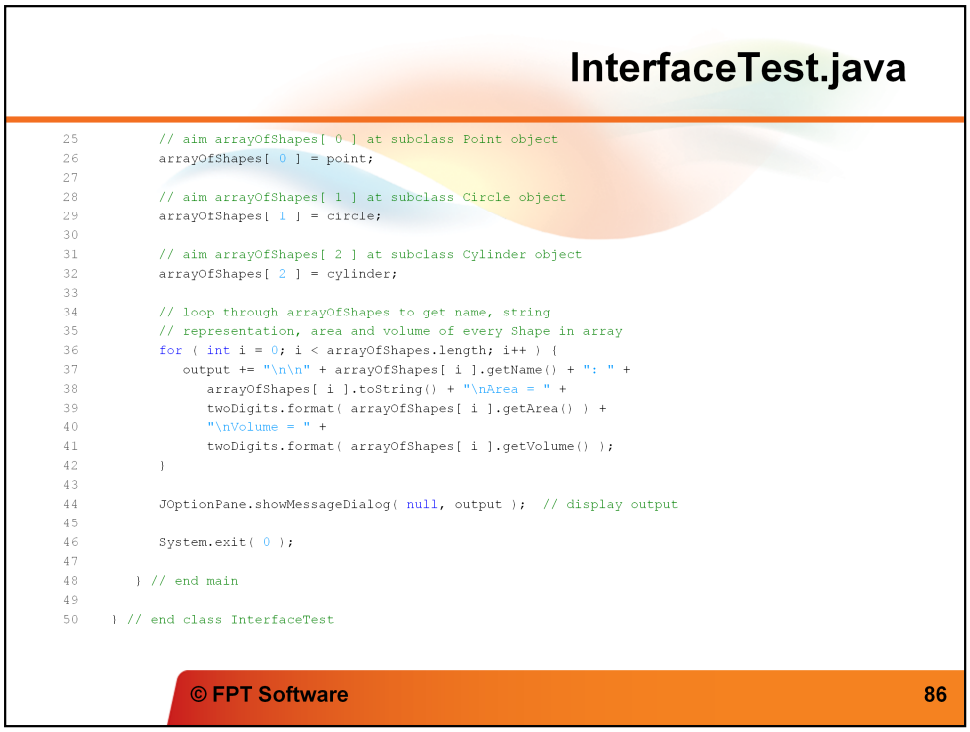


Lines 47-59: Implement methods specified by interface `Shape`

InterfaceTest.java

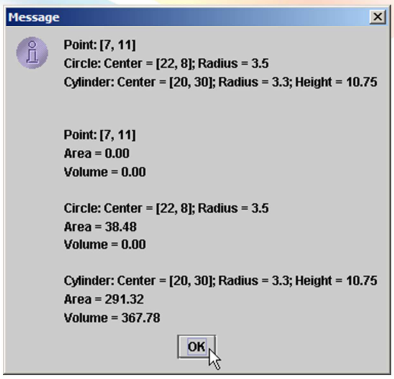
```
1   // InterfaceTest.java
2   // Test Point, Circle, Cylinder hierarchy with interface Shape.
3   import java.text.DecimalFormat;
4   import javax.swing.JOptionPane;
5
6   public class InterfaceTest {
7
8      public static void main( String args[] )
9      {
10       // set floating-point number format
11       DecimalFormat twoDigits = new DecimalFormat( "0.00" );
12
13       // create Point, Circle and Cylinder objects
14       Point point = new Point( 7, 11 );
15       Circle circle = new Circle( 22, 8, 3.5 );
16       Cylinder cylinder = new Cylinder( 20, 30, 3.3, 10.75 );
17
18       // obtain name and string representation of each object
19       String output = point.getName() + ": " + point + "\n" +
20          circle.getName() + ": " + circle + "\n" +
21          cylinder.getName() + ": " + cylinder + "\n";
22
23       Shape arrayOfShapes[] = new Shape[ 3 ];  // create Shape array
24
```


Line 23: Create `Shape` array



Lines 36-42: Loop through `arrayOfShapes` to get name, string representation, area and volume of every shape in array.

Output


Message
Point: [7, 11]
Circle: Center = [22, 8]; Radius = 3.5
Cylinder: Center = [20, 30]; Radius = 3.3; Height = 10.75
Point: [7, 11]
Area = 0.00
Volume = 0.00
Circle: Center = [22, 8]; Radius = 3.5
Area = 38.48
Volume = 0.00
Cylinder: Center = [20, 30]; Radius = 3.3; Height = 10.75
Area = 291.32
Volume = 367.78
OK

Overriding

- An instance method in a subclass with the same signature (name, plus the number and the type of its parameters) and return type as an instance method in the superclass *overrides* the superclass's method.
- The ability of a subclass to override a method allows a class to inherit from a superclass whose behavior is "close enough" and then to modify behavior as needed.
- The overriding method has the same name, number and type of parameters, and return type as the method it overrides.
- An overriding method can also return a subtype of the type returned by the overridden method. This is called a *covariant return type*.

User Shape, Point, Circle and Cylinder from AbstractInheritanceTest to show example

Overriding

- The access modifier for an overriding method can allow more, but not less, access than the overridden method. For example, a protected instance method in the superclass can be made public, but not private, in the subclass.
- You will get a compile-time error if you attempt to change an instance method in the superclass to a class method in the subclass, and vice versa.
- You cannot override a method marked final.

User Shape, Point, Circle and Cylinder from AbstractInheritanceTest to show example

Hiding method

- If a subclass defines a **class method** with the same signature as a **class method** in the superclass, the method in the subclass *hides* the one in the superclass.

Overriding and Hiding Example

```
public class Shape {
    public static void testClassMethod() {
        System.out.println("The class method in Shape.");
    }

    public void testInstanceMethod() {
        System.out.println("The instance method in Shape.");
    }
}

public class Circle extends Shape {
    public static void testClassMethod() {
        System.out.println("The class method in Circle.");
    }

    public void testInstanceMethod() {
        System.out.println("The instance method in Circle.");
    }
}
```


The first is Shape, which contains one instance method and one class method.

The second class, a subclass of Shape, is called Circle. The Circle class overrides the instance method in Shape and hides the class method in Shape.

Overriding and Hiding Example

```
public class TestOverridingAndHiding {
    public static void main(String[] args) {
        Circle myCircle = new Circle();
        Shape myShape = myCircle;

        Shape.testClassMethod();
        myShape.testInstanceMethod();
    }
}
```

Output:

```
The class method in Shape.
The instance method in Circle.
```


As promised, the version of the hidden method that gets invoked is the one in the superclass, and the version of the overridden method that gets invoked is the one in the subclass.

Overloading

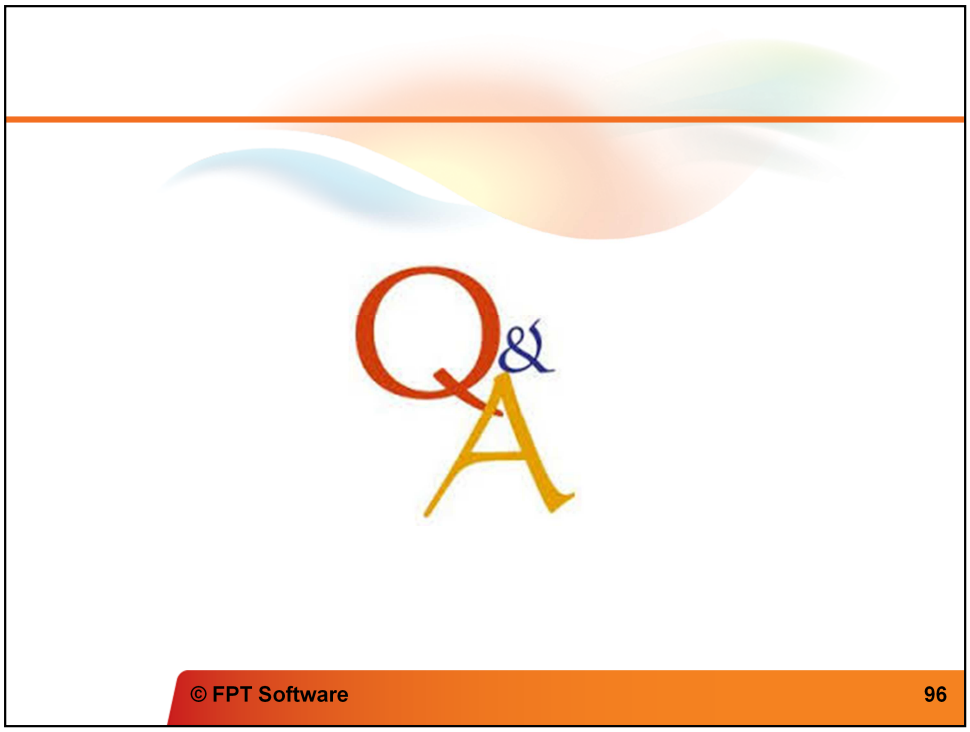
- Overloaded methods let you reuse the same method name in a class, but with different arguments (and optionally, a different return type).
- In a subclass, you can overload the methods inherited from the superclass. Such overloaded methods neither hide nor override the superclass methods—they are new methods, unique to the subclass.

Overloading

- Some main rules for overloading a method:
  - Overloaded methods **must** change the argument list.
  - Overloaded methods can change the return type.
  - Overloaded methods can change the access modifier.
  - A method can be overloaded in the same class or in a subclass.

Summary

- **Polymorphism**, which means "many forms," is the ability to treat an object of any subclass of a base class as if it were an object of the base class.
- **Abstract class** is a class that may contain abstract methods and implemented methods. An *abstract* method is one without a body that is declared with the reserved word *abstract*
- An **interface** is a collection of constants and method declarations. When a class *implements* an interface, it must declare and provide a method body for each method in the interface


Q&A
© FPT Software
96