Inheritance

Inheritance Motivation

- Inheritance in Java is achieved through extending classes

Inheritance enables:

- Code re-use
- Grouping similar code
- Flexibility to customize

Inheritance Concepts

- Many real-life objects are related in a hierarchical fashion such that lower levels of the hierarchy inherit characteristics of the upper levels.

  e.g.,

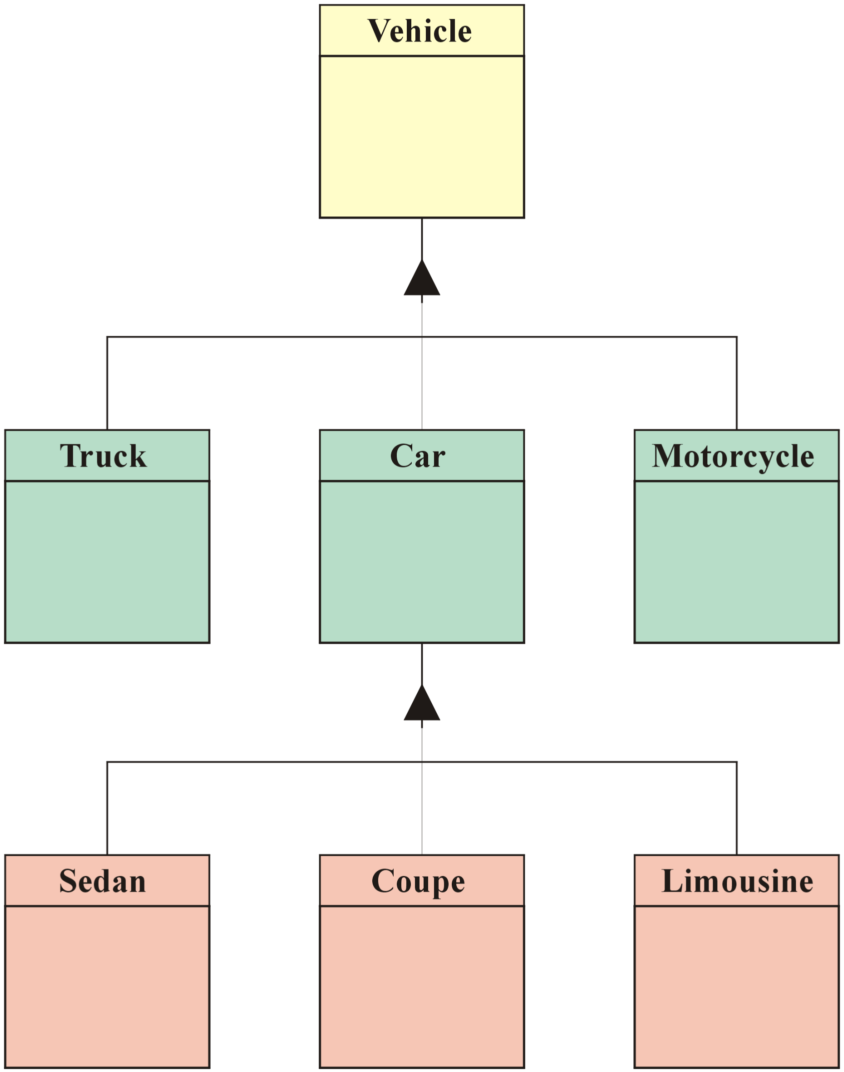
  mammal ⇒ primate ⇒ human

  vehicle ⇒ car ⇒ Honda Accord

  person ⇒ employee ⇒ faculty member
- These types of hierarchies/relationships may be called **IS-A** (e.g., a primate **is-a** mammal).

Inheritance Concepts - Hierarchy

- The inheritance hierarchy is usually drawn as an inverted (upside-down) tree.
- The tree can have any number of levels.
- The class at the top (base) of the inverted tree is called the ***root class***.
- In Java, the root class is called ***Object***.



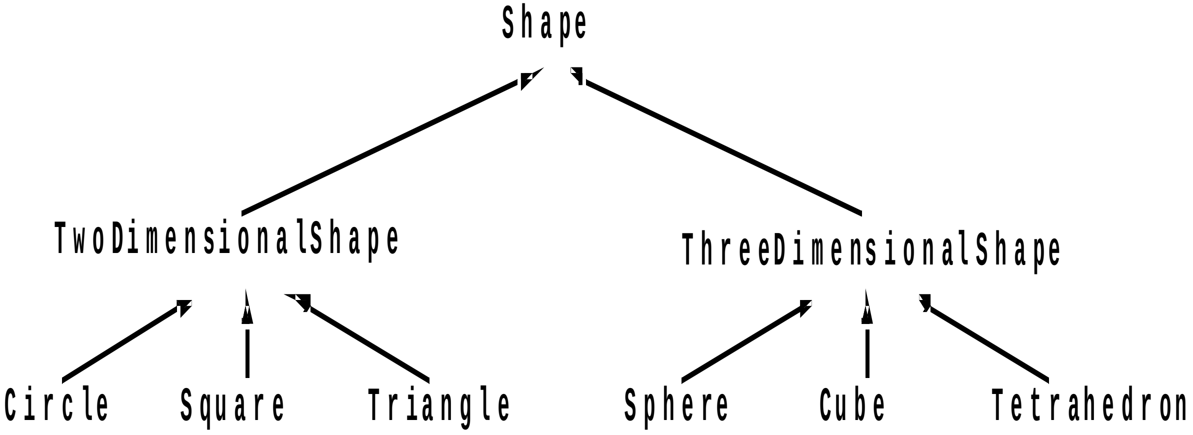
Object Root Class

- The Object root class provides the following capabilities to all Java objects:
  - Event handling - synchronizing execution of multiple executable objects (e.g., a print spooler and a printer driver)
  - Cloning – creating an exact copy of an object
  - Finalization – cleaning up when an object is no longer needed
  - Equality checking
  - Querying runtime class

Inheritance Concepts - Terms

- OOP languages provide specific mechanisms for defining inheritance relationships between classes.
- ***Derived (Child) Class*** - a class that inherits characteristics of another class.
- ***Base (Parent) Class*** - a class from which characteristics are inherited by one or more other classes.
- A derived class inherits data and function members from **ALL** of its base classes.

A portion of a `Shape` class hierarchy.



Shape

TwoDimensionalShape

ThreeDimensionalShape

Circle

Square

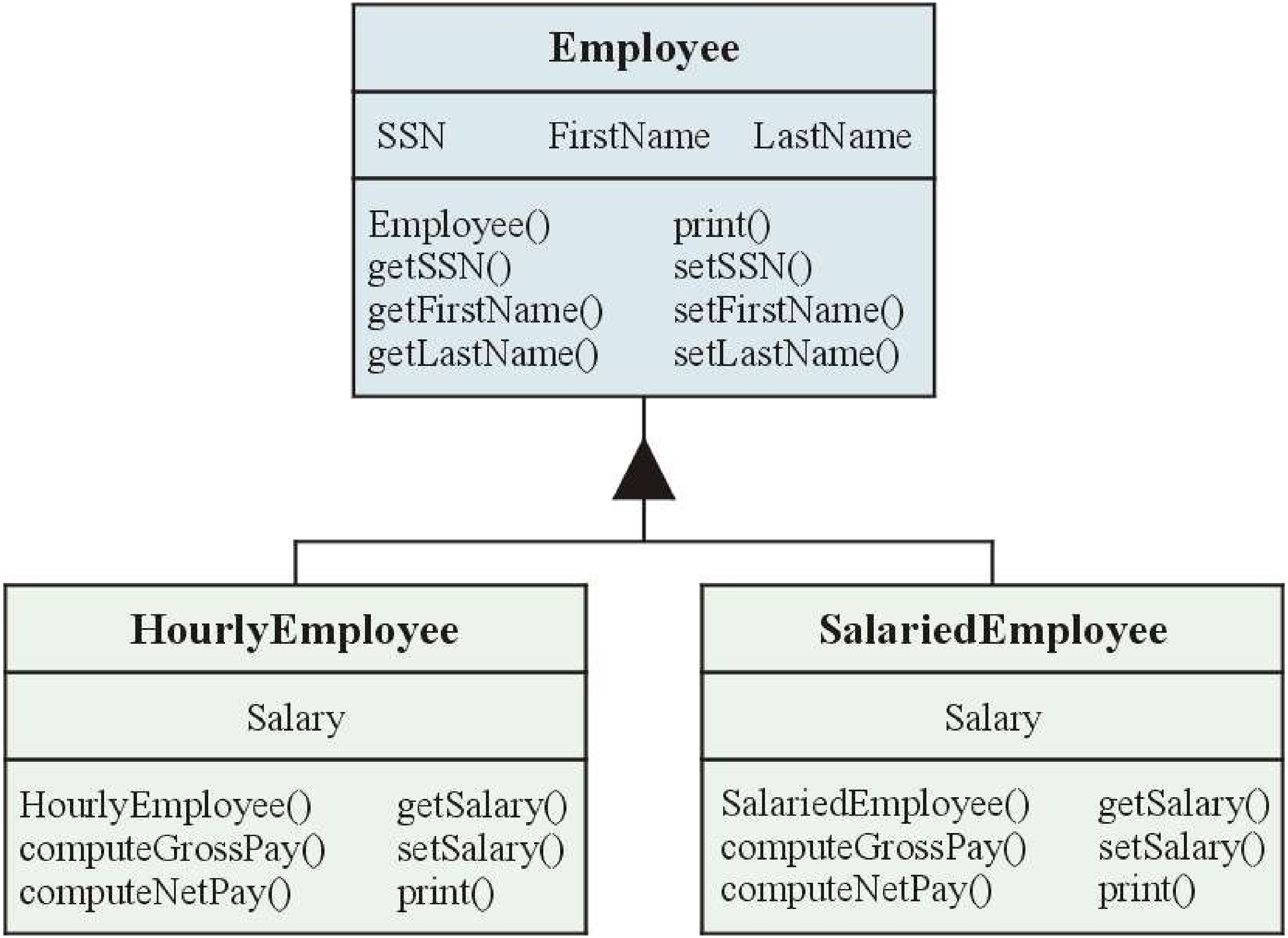
Triangle

Sphere

Cube

Tetrahedron

Inheritance Concepts - Example



Employee

SSN FirstName LastName

Employee() print()
getSSN() setSSN()
getFirstName() setFirstName()
getLastName() setLastName()

HourlyEmployee

Salary

HourlyEmployee() getSalary()
computeGrossPay() setSalary()
computeNetPay() print()

SalariedEmployee

Salary

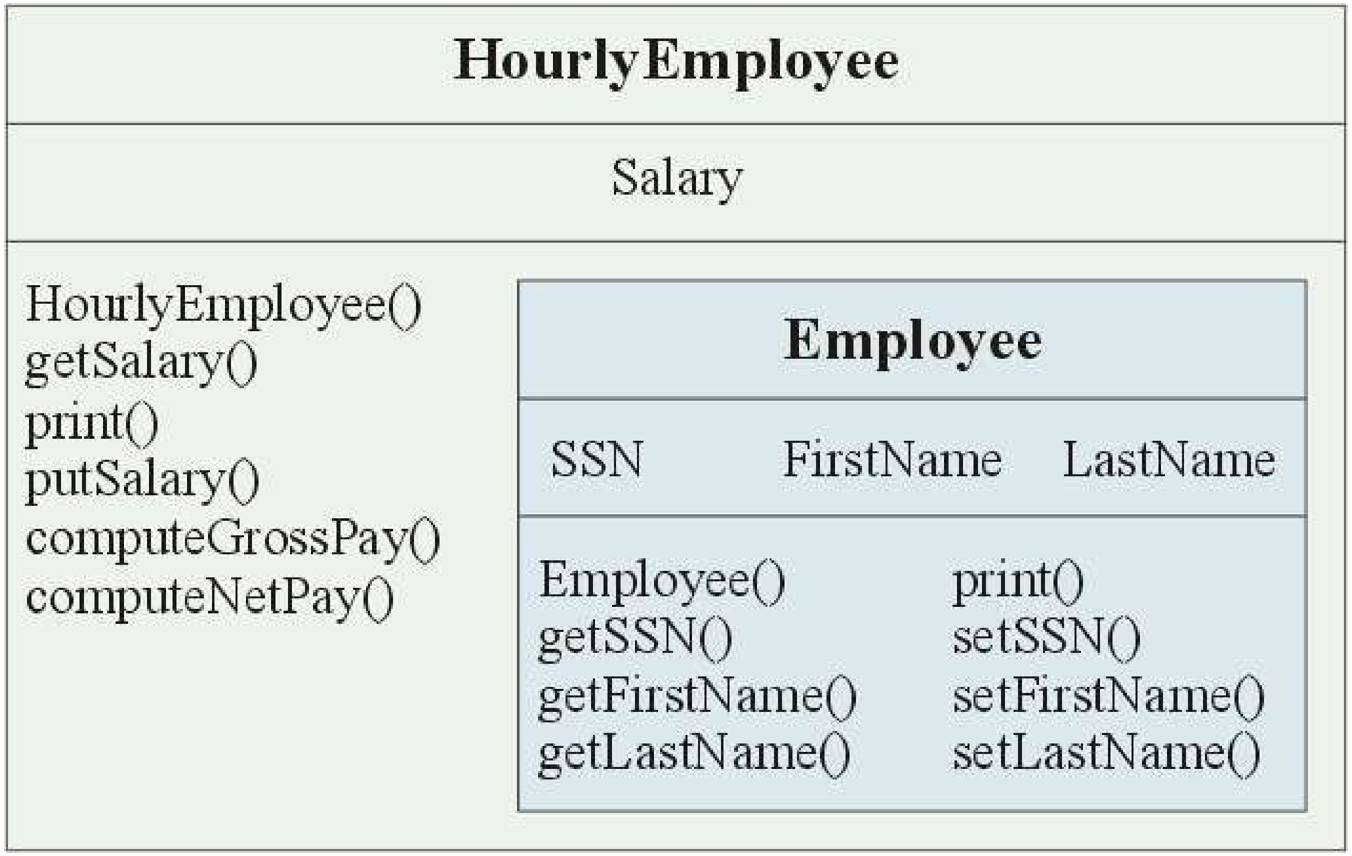
SalariedEmployee() getSalary()
computeGrossPay() setSalary()
computeNetPay() print()

Inheritance Concepts – Employee Example

- Employee is a base class.
- HourlyEmployee and SalariedEmployee are derived classes that inherit all data and function members from Employee.
  - e.g., SSN and setLastName()
- Each derived class can add data and function members.
  - e.g., Salary and computeGrossPay())
- Different derived classes can defined the same members with different data types or implementations.
  - e.g., Salary can be a float in HourlyEmployee and an int in SalariedEmployee, computeGrossPay() can use a different algorithm in each derived class.

Inheritance - Embedded Objects

- Think of each derived class object as having a base class object embedded within it.



Java Inheritance Declarations

- No special coding is required to designate a base class, e.g.,

```
class Employee { … }
```

- A derived class must specifically declare the base class from which it is derived, e.g.,

```
class HourlyEmployee extends Employee { … }
```

Abstract and Final Classes

- A base class can be declared with the keyword ***abstract***, e.g.,

```
abstract class Employee { … }
```

- An abstract class cannot be instantiated
  - An attempt to create an object of type Employee generates an InstantiationError exception.
- A class can be declared with the keyword ***final***, e.g.,

```
final class Employee { … }
```

- A final class cannot be extended

Inheritance - Constructor Functions

- When an object of a derived class is created, the constructor functions of the derived class and all base classes are also called.
  - In what order are constructor functions called and executed?
  - How are parameters passed to base class constructor functions?

Constructor Function Calls

- The derived class constructor function is called first - it instantiates its parameters then calls the base class constructor function.
- The base class constructor function instantiates its parameters and calls its base class constructor function.
- If a base class has no base classes, it executes the body of its constructor function
- When a base class constructor function terminates the nearest derived class constructor function executes.
- Calls and parameter instantiation go “up the hierarchy”, execution goes “down the hierarchy”

Constructor Function Calls - Example

Order of call and execution for HourlyEmployee():

- HourlyEmployee() is called
- HourlyEmployee() instantiates its parameters and calls Employee()
- Employee() instantiates its parameters and calls Object()
- Object() instantiates its parameters, executes, and returns to Employee()
- Employee() executes and returns to HourlyEmployee ()
- HourlyEmployee() executes and returns

The keyword super

- Derived classes often need a way to refer to data and function members in the base class.

- The keyword ***super***, used within a derived class function, is an alias for the name of the base class.

- In the Employee example, the compiler automatically substitutes “Employee” for “super” anywhere it appears within member functions of HourlyEmployee and SalariedEmployee.

Constructor Function Parameters

- When an object is created:
  - only one constructor function is called
  - parameters are passed only to that constructor function
- For example, the following code creates an HourlyEmployee object and passes all parameters to the HourlyEmployee constructor function:

```
HourlyEmployee Tom;
Tom = new HourlyEmployee(123456789,"Tom","Jones",15.50f);
```

- But the first three parameters “belong” to the base class (Employee) constructor function.
- How can those parameters be passed to the Employee constructor function?

Constructor Parameters - Continued

- The keyword **super** enables parameter passing between derived and base class constructor functions
- For example, the following code passes the first three parameters from the HourlyEmployee constructor function to the Employee constructor function:

```
public HourlyEmployee(int newSSN, String newFirstName,
                        String newLastName, float
newSalary)
{
    super(newSSN,newFirstName,newLastName);
    salary=newSalary;
} // end HourlyEmployee(int,String,String,float)
```

Constructor Parameters - Continued

- If a **super** call to the base class constructor function appears in a derived class constructor function <u>it must be the first executable line</u>.
- If there are no parameters to pass to the base class constructor function, the **super** call can be omitted.
- If the **super** call is omitted, the compiler automatically inserts a call to the base class default constructor function at the beginning of the derived class constructor function.

Inheritance - Member Override

- When a member of a derived class has the same name as a member of a base class the derived class member is said to ***override*** the base class member.
- The derived class member “hides” the base class member unless a qualified name is used.
- Both the base and derived class implementations of the overridden data or function member exist within each derived class object.

Naming Ambiguity

- Inheritance creates naming ambiguities when both base and derived classes have data or function members of the same name.
- Consider print() in the Employee example – both the base class Employee and the derived classes HourlyEmployee and Salaried employee declare print()
- In the following code, which version of print() is called?

```
HourlyEmployee Jane = new HourlyEmployee();
Jane.print();
```

Naming Ambiguity - Continued

- In the previous example the version of print() defined in HourlyEmployee() will always be called, if it exists.
- If there is no print() in HourlyEmployee, the compiler will look for it in the parent class Employee.
- The compiler searches up the inheritance hierarchy until it finds print() or runs out of parent classes (at which point it will generate an error).

Naming Ambiguity - Continued

- Naming ambiguity can also exist within derived class functions. For example, what version of print() is called in the following code?

```
public class HourlyEmployee extends Employee
{
   // … data and other function declarations not shown

   public void print()
   {
       print();
       System.out.print(“salary=“);
       System.out.println(salary);
   }
} // end class HourlyEmployee
```

Naming Ambiguity - Rules

- The embedded call to print() on the previous slide is an example of an ***unqualified name*** or reference (it’s also a recursive function call!).
- It is “unqualified” because the programmer hasn’t explicitly told the compiler which print() version to call.
- The compiler always resolves an unqualified name to the “closest” instance - defined as follows:
  1. A local name (e.g., a function parameter)
  2. A member of the derived class
  3. A member of the closest base class

Qualifying Names With super

- A name can be qualified with the keyword super to override the default resolution.
- For example, the statement:

```
super.print();
```

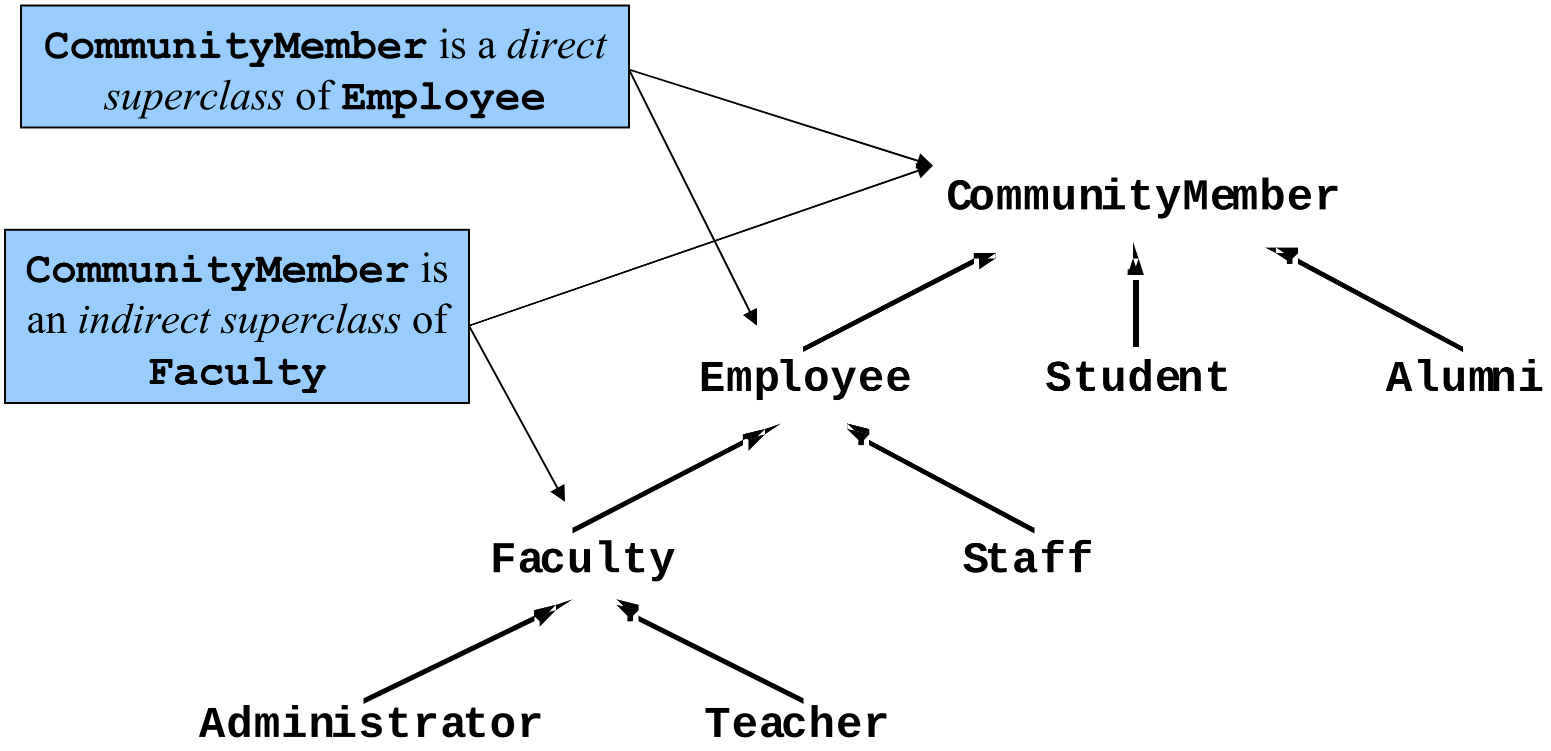
within a derived class function explicitly calls the base class print() function.

More Example

Superclass	Subclasses
Student	**GraduateStudent** **UndergraduateStudent**
Shape	**Circle** **Triangle** **Rectangle**
Loan	**CarLoan** **HomeImprovementLoan** **MortgageLoan**
Employee	**FacultyMember** **StaffMember**
Account	**CheckingAccount** **SavingsAccount**

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

Fig. 9.2 An inheritance hierarchy for university CommunityMembers.



CommunityMember is a *direct superclass* of **Employee**

CommunityMember is an *indirect superclass* of **Faculty**

CommunityMember

Employee

Student

Alumni

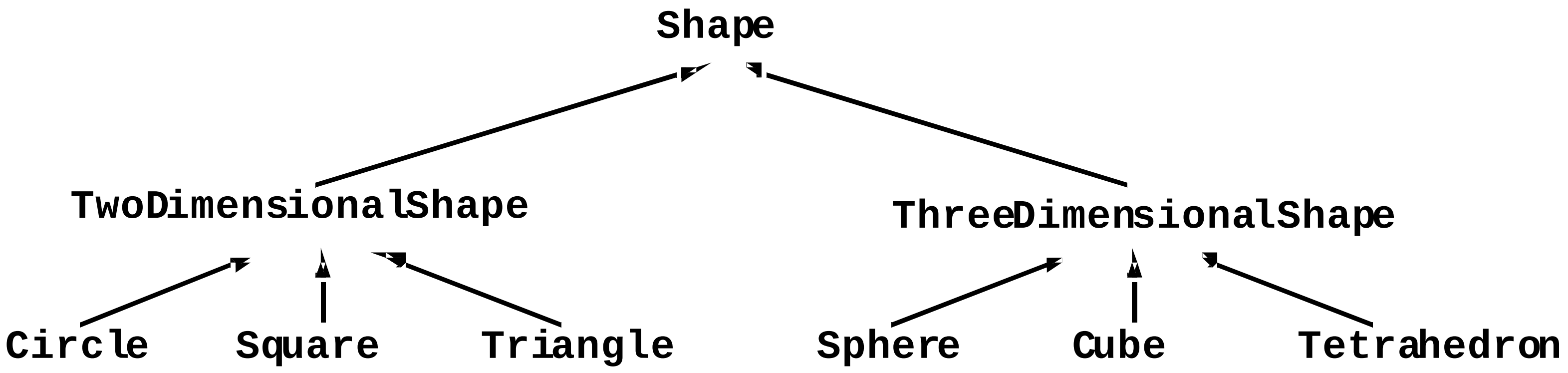
Faculty

Staff

Administrator

Teacher

Fig. 9.3 A portion of a Shape class hierarchy.



protected Members

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

```
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)

`Point.java`

Line 5

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

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

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

}  // end class Point
```

Point.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 superclas
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**

Circle.java

Line 4
Circle is a **Point** subclass

Line 4
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 Circl
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**

InheritanceTest.java

Lines 18-19
Instantiate objects

Line 22
Circle invokes method **toString**

s object
references subclass

9
t invokes
Circle’s **toString**
method

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

```
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**

InheritanceTest.

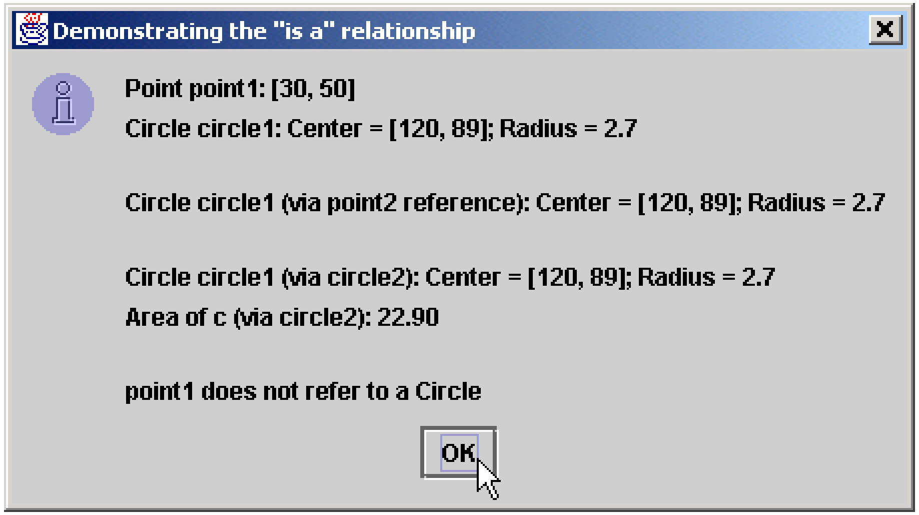
Line 36
nvokes its
n **toString**
method

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



```
Point point1: [30, 50]
Circle circle1: Center = [120, 89]; Radius = 2.7

Circle circle1 (via point2 reference): Center = [120, 89]; Radius = 2.7

Circle circle1 (via circle2): Center = [120, 89]; Radius = 2.7
Area of c (via circle2): 22.90

point1 does not refer to a Circle
```

9.5 Constructors and Finalizers in Subclasses (cont.)

- **`finalize`** method
  - Garbage collection
  - Subclass **`finalize`** method
    - should invoke superclass **`finalize`** method

```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 final
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

Point.java

Lines 7-20
Superclass constructors

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

```
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 final
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**

ircle.java

Line 9
Implicit call to **Point** constructor

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

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

```
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**

Test.java

Lines 10-11
Instantiate **Circle** objects

Line 17
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
```