9

Object-Oriented Programming: Inheritance

Say not you know another entirely, till you have divided an inheritance with him.
—Johann Kasper Lavater

This method is to define as the number of a class the class of all classes similar to the given class.
—Bertrand Russell

Good as it is to inherit a library, it is better to collect one.
—Augustine Birrell

Save base authority from others' books.
—William Shakespeare

OBJECTIVES

In this chapter you will learn:

- How inheritance promotes software reusability.
- The notions of superclasses and subclasses.
- To use keyword `extends` to create a class that inherits attributes and behaviors from another class.
- To use access modifier `protected` to give subclass methods access to superclass members.
- To access superclass members with `super`.
- How constructors are used in inheritance hierarchies.
- The methods of class `Object`, the direct or indirect superclass of all classes in Java.

Student Solution Exercises

9.3 Many programs written with inheritance could be written with composition instead, and vice versa. Rewrite class `BasePlusCommissionEmployee4` (Fig. 9.13) of the `CommissionEmployee3–BasePlusCommissionEmployee4` hierarchy to use composition rather than inheritance. After you do this, assess the relative merits of the two approaches for the `CommissionEmployee3` and `BasePlusCommissionEmployee4` problems, as well as for object-oriented programs in general. Which approach is more natural? Why?

ANS: For a relatively short program like this one, either approach is acceptable. But as programs become larger with more and more objects being instantiated, inheritance becomes preferable because it makes the program easier to modify and promotes the reuse of code. The inheritance approach is more natural because a base-salaried commission employee *is a* commission employee. Composition is defined by the "has-a" relationship, and clearly it would be strange to say that "a base-salaried commission employee *has a* commission employee."

```
1  // Exercise 9.3 Solution: BasePlusCommissionEmployee4.java
2  // BasePlusCommissionEmployee4 using composition.
3
4  public class BasePlusCommissionEmployee4
5  {
6     private CommissionEmployee3 commissionEmployee; // composition
7     private double baseSalary; // base salary per week
8
9     // six-argument constructor
10    public BasePlusCommissionEmployee4( String first, String last,
11       String ssn, double sales, double rate, double salary )
12    {
13       commissionEmployee =
14          new CommissionEmployee3( first, last, ssn, sales, rate );
15       setBaseSalary( salary ); // validate and store base salary
16    } // end six-argument BasePlusCommissionEmployee4 constructor
17
18    // set first name
19    public void setFirstName( String first )
20    {
21       commissionEmployee.setFirstName( first );
22    } // end method setFirstName
23
24    // return first name
25    public String getFirstName()
26    {
27       return commissionEmployee.getFirstName();
28    } // end method getFirstName
29
30    // set last name
31    public void setLastName( String last )
32    {
33       commissionEmployee.setLastName( last );
34    } // end method setLastName
35
36    // return last name
37    public String getLastName()
38    {
```

```
39          return commissionEmployee.getLastName();
40       } // end method getLastName
41
42       // set social security number
43       public void setSocialSecurityNumber( String ssn )
44       {
45          commissionEmployee.setSocialSecurityNumber( ssn );
46       } // end method setSocialSecurityNumber
47
48       // return social security number
49       public String getSocialSecurityNumber()
50       {
51          return commissionEmployee.getSocialSecurityNumber();
52       } // end method getSocialSecurityNumber
53
54       // set commission employee's gross sales amount
55       public void setGrossSales( double sales )
56       {
57          commissionEmployee.setGrossSales( sales );
58       } // end method setGrossSales
59
60       // return commission employee's gross sales amount
61       public double getGrossSales()
62       {
63          return commissionEmployee.getGrossSales();
64       } // end method getGrossSales
65
66       // set commission employee's rate
67       public void setCommissionRate( double rate )
68       {
69          commissionEmployee.setCommissionRate( rate );
70       } // end method setCommissionRate
71
72       // return commission employee's rate
73       public double getCommissionRate()
74       {
75          return commissionEmployee.getCommissionRate();
76       } // end method getCommissionRate
77
78       // set base-salaried commission employee's base salary
79       public void setBaseSalary( double salary )
80       {
81          baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
82       } // end method setBaseSalary
83
84       // return base-salaried commission employee's base salary
85       public double getBaseSalary()
86       {
87          return baseSalary;
88       } // end method getBaseSalary
89
90       // calculate base-salaried commission employee's earnings
91       public double earnings()
92       {
93          return getBaseSalary() + commissionEmployee.earnings();
```

```
94      } // end method earnings
95
96      // return String representation of BasePlusCommissionEmployee4
97      public String toString()
98      {
99         return String.format( "%s %s\n%s: %.2f", "base-salaried",
100           commissionEmployee.toString(), "base salary", getBaseSalary() );
101     } // end method toString
102  } // end class BasePlusCommissionEmployee4
```

9.5 Draw an inheritance hierarchy for students at a university similar to the hierarchy shown in Fig. 9.2. Use `Student` as the superclass of the hierarchy, then extend `Student` with classes `UndergraduateStudent` and `GraduateStudent`. Continue to extend the hierarchy as deep (i.e., as many levels) as possible. For example, `Freshman`, `Sophomore`, `Junior` and `Senior` might extend `UndergraduateStudent`, and `DoctoralStudent` and `MastersStudent` might be subclasses of `GraduateStudent`. After drawing the hierarchy, discuss the relationships that exist between the classes. [*Note:* You do not need to write any code for this exercise.]

ANS:

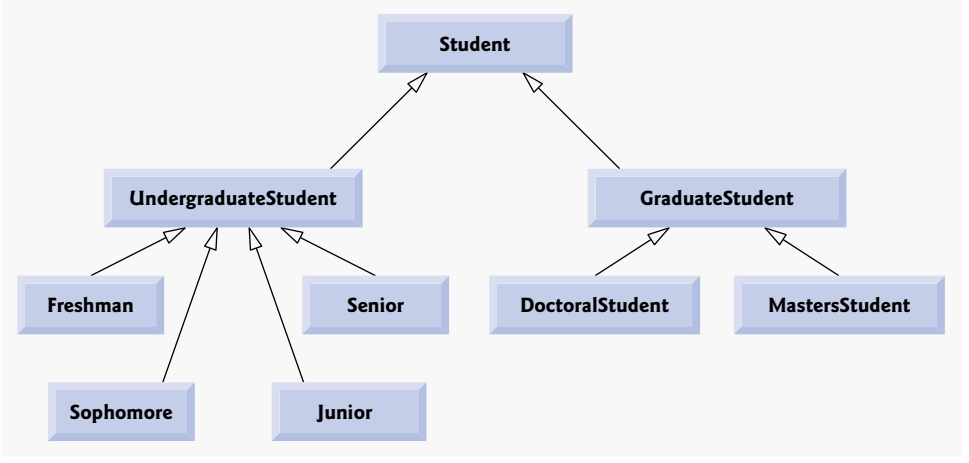

Fig. 9.1 |

This hierarchy contains many "is-a" (inheritance) relationships. An `UndergraduateStudent` *is a* `Student`. A `GraduateStudent` *is a* `Student` too. Each of the classes `Freshman`, `Sophomore`, `Junior` and `Senior` *is an* `UndergraduateStudent` and *is a* `Student`. Each of the classes `DoctoralStudent` and `MastersStudent` *is a* `GraduateStudent` and *is a* `Student`.

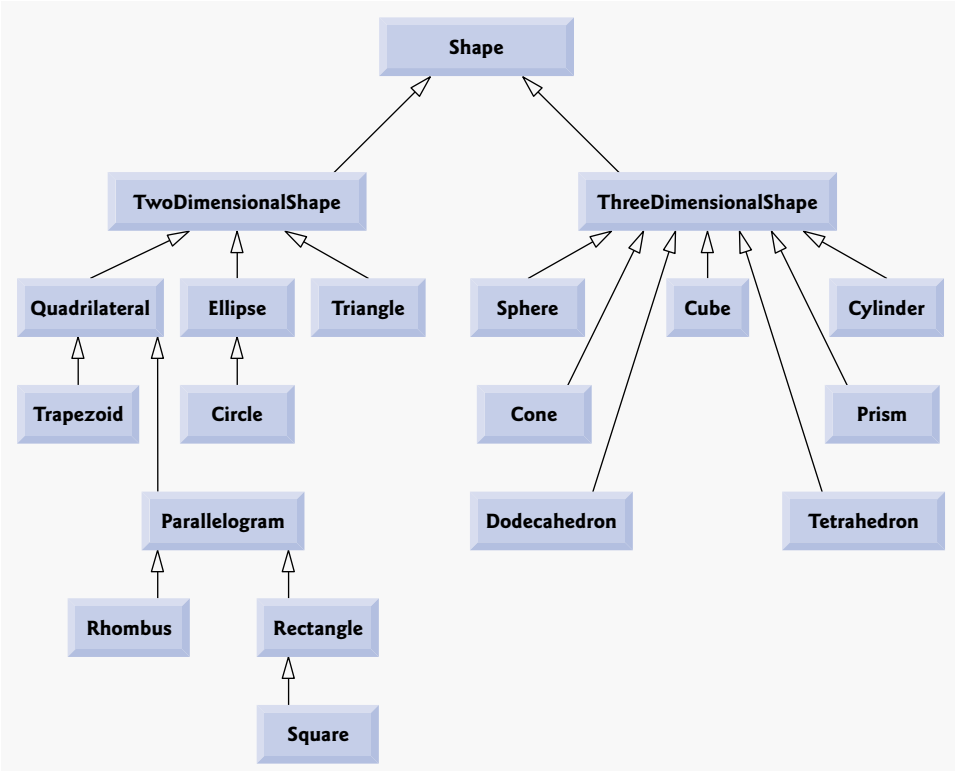

Fig. 9.2 |

9.7 Some programmers prefer not to use `protected` access, because they believe it breaks the encapsulation of the superclass. Discuss the relative merits of using `protected` access vs. using `private` access in superclasses.

ANS: `private` instance variables are hidden in the subclass and are accessible only through the `public` or `protected` methods of the superclass. Using `protected` access enables the subclass to manipulate the `protected` members without using the access methods of the superclass. If the superclass members are `private`, the methods of the superclass must be used to access the data. This may result in a decrease in performance due to the extra method calls.