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

Objectives

In this chapter you'll:

- Understand inheritance and how to use it to develop new classes based on existing classes.
- Learn the notions of superclasses and subclasses and the relationship between them.
- Use keyword `extends` to create a class that inherits attributes and behaviors from another class.
- Use access modifier `protected` in a superclass to give subclass methods access to these superclass members.
- Access superclass members with `super` from a subclass.
- Learn how constructors are used in inheritance hierarchies.
- Learn about the methods of class `Object`, the direct or indirect superclass of all classes.

Outline

9.1 Introduction
9.2 Superclasses and Subclasses
9.3 `protected` Members
9.4 Relationship Between Superclasses and Subclasses
9.4.1 Creating and Using a `CommissionEmployee` Class
9.4.2 Creating and Using a `BasePlusCommissionEmployee` Class
9.4.3 Creating a `CommissionEmployee–BasePlusCommissionEmployee` Inheritance Hierarchy
9.4.4 `CommissionEmployee–BasePlusCommissionEmployee` Inheritance Hierarchy Using `protected` Instance Variables
9.4.5 `CommissionEmployee–BasePlusCommissionEmployee` Inheritance Hierarchy Using `private` Instance Variables
9.5 Constructors in Subclasses
9.6 Class `Object`
9.7 (Optional) GUI and Graphics Case Study: Displaying Text and Images Using `Label`s
9.8 Wrap-Up

Summary | *Self-Review Exercises* | *Answers to Self-Review Exercises* | *Exercises*

9.1 Introduction

This chapter continues our discussion of object-oriented programming (OOP) by introducing **inheritance**, in which a new class is created by acquiring an existing class's members and possibly embellishing them with new or modified capabilities. With inheritance, you can save time during program development by basing new classes on existing proven and debugged high-quality software. This also increases the likelihood that a system will be implemented and maintained effectively.

When creating a class, rather than declaring completely new members, you can designate that the new class should *inherit* the members of an existing class. The existing class is called the **superclass**, and the new class is the **subclass.** (The C++ programming language refers to the superclass as the **base class** and the subclass as the **derived class**.) A subclass can become a superclass for future subclasses.

A subclass can add its own fields and methods. Therefore, a subclass is *more specific* than its superclass and represents a more specialized group of objects. The subclass exhibits the behaviors of its superclass and can modify those behaviors so that they operate appropriately for the subclass. This is why inheritance is sometimes referred to as **specialization.**

The **direct superclass** is the superclass from which the subclass explicitly inherits. An **indirect superclass** is any class above the direct superclass in the **class hierarchy**, which defines the inheritance relationships among classes—as you'll see in Section 9.2, diagrams help you understand these relationships. In Java, the class hierarchy begins with class `Object` (in package `java.lang`), which *every* class in Java directly or indirectly **extends** (or "inherits from"). Section 9.6 lists the methods of class `Object` that are inherited by all other Java classes. Java supports only **single inheritance**, in which each class is derived from exactly *one* direct superclass. Unlike C++, Java does *not* support multiple inheritance (which occurs when a class is derived from more than one direct superclass). Chapter 10, Object-Oriented Programming: Polymorphism and Interfaces, explains how to use Java *interfaces* to realize many of the benefits of multiple inheritance while avoiding the associated problems.

We distinguish between the ***is-a* relationship** and the ***has-a* relationship**. *Is-a* represents inheritance. In an *is-a* relationship, *an object of a subclass can also be treated as an object of its superclass*—e.g., a car *is a* vehicle. By contrast, *has-a* represents composition (see Chapter 8). In a *has-a* relationship, *an object contains as members references to other objects*—e.g., a car *has a* steering wheel (and a car object has a reference to a steering-wheel object).

New classes can inherit from classes in **class libraries**. Organizations develop their own class libraries and can take advantage of others available worldwide. Some day, most new software likely will be constructed from **standardized reusable components**, just as automobiles and most computer hardware are constructed today. This will facilitate the rapid development of more powerful, abundant and economical software.

9.2 Superclasses and Subclasses

Often, an object of one class *is an* object of another class as well. For example, a `CarLoan` *is a* `Loan` as are `HomeImprovementLoans` and `MortgageLoans`. Thus, in Java, class `CarLoan` can be said to inherit from class `Loan`. In this context, class `Loan` is a superclass and class `CarLoan` is a subclass. A `CarLoan` *is a* specific type of `Loan`, but it's incorrect to claim that every `Loan` *is a* `CarLoan`—the `Loan` could be any type of loan. Figure 9.1 lists several simple examples of superclasses and subclasses—superclasses tend to be "more general" and subclasses "more specific."

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

Fig. 9.1 | Inheritance examples.

Because every subclass object *is an* object of its superclass, and one superclass can have many subclasses, the set of objects represented by a superclass is often larger than the set of objects represented by any of its subclasses. For example, the superclass `Vehicle` represents *all* vehicles, including cars, trucks, boats, bicycles and so on. By contrast, subclass `Car` represents a smaller, more specific subset of vehicles.

University Community Member Hierarchy

Inheritance relationships form treelike *hierarchical* structures. A superclass exists in a hierarchical relationship with its subclasses. Let's develop a sample class hierarchy (Fig. 9.2), also called an **inheritance hierarchy**. A university community has thousands of members, including employees, students and alumni. Employees are either faculty or staff members. Faculty members are either administrators (e.g., deans and department chairpersons) or teachers. The hierarchy could contain many other classes. For example, students can be graduate or undergraduate students. Undergraduate students can be freshmen, sophomores, juniors or seniors.

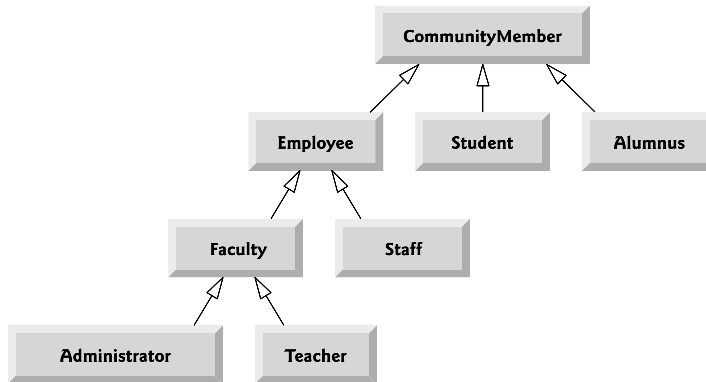

Fig. 9.2 | Inheritance hierarchy UML class diagram for university `CommunityMember`s.

Each arrow in the hierarchy represents an *is-a* relationship. As we follow the arrows upward in this class hierarchy, we can state, for example, that "an `Employee` *is a* `CommunityMember`" and "a `Teacher` *is a* `Faculty` member." `CommunityMember` is the direct superclass of `Employee`, `Student` and `Alumnus` and is an indirect superclass of all the other classes in the diagram. Starting from the bottom, you can follow the arrows and apply the *is-a* relationship up to the topmost superclass. For example, an `Administrator` *is a* `Faculty` member, *is an* `Employee`, *is a* `CommunityMember` and, of course, *is an* `Object`.

Shape Hierarchy

Now consider the `Shape` inheritance hierarchy in Fig. 9.3. This hierarchy begins with superclass `Shape`, which is extended by subclasses `TwoDimensionalShape` and `ThreeDimensionalShape`—`Shape`s are either `TwoDimensionalShape`s or `ThreeDimensionalShape`s. The third level of this hierarchy contains *specific* types of `TwoDimensionalShape`s and `ThreeDimensionalShape`s. As in Fig. 9.2, we can follow the arrows from the bottom of the diagram to the topmost superclass in this class hierarchy to identify several *is-a* relationships. For example, a `Triangle` *is a* `TwoDimensionalShape` and *is a* `Shape`, while a `Sphere` *is a* `ThreeDimensionalShape` and *is a* `Shape`. This hierarchy could contain many other classes. For example, ellipses and trapezoids also are `TwoDimensionalShape`s.

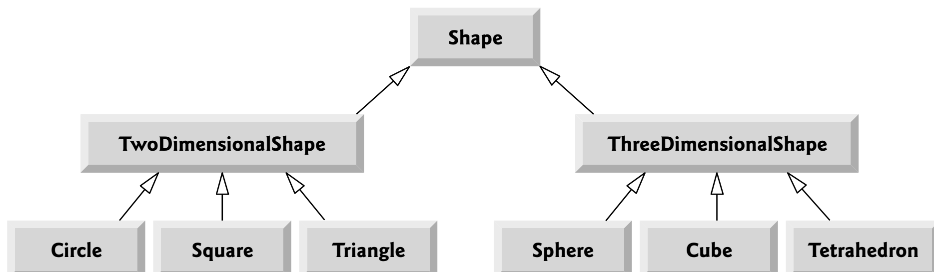

Fig. 9.3 | Inheritance hierarchy UML class diagram for `Shape`s.

Not every class relationship is an inheritance relationship. In Chapter 8, we discussed the *has-a* relationship, in which classes have members that are references to objects of other classes. Such relationships create classes by *composition* of existing classes. For example, given the classes `Employee`, `BirthDate` and `TelephoneNumber`, it's improper to say that an `Employee` *is a* `BirthDate` or that an `Employee` *is a* `TelephoneNumber`. However, an `Employee` *has a* `BirthDate`, and an `Employee` *has a* `TelephoneNumber`.

It's possible to treat superclass objects and subclass objects similarly—their commonalities are expressed in the superclass's members. Objects of all classes that extend a common superclass can be treated as objects of that superclass—such objects have an *is-a* relationship with the superclass. Later in this chapter and in Chapter 10, we consider many examples that take advantage of the *is-a* relationship.

A subclass can customize methods that it inherits from its superclass. To do this, the subclass **overrides** (*redefines*) the superclass method with an appropriate implementation, as we'll see in the chapter's code examples.

9.3 protected Members

Chapter 8 discussed access modifiers `public` and `private`. A class's `public` members are accessible wherever the program has a *reference* to an *object* of that class or one of its *subclasses*. A class's `private` members are accessible only within the class itself. In this section, we introduce the access modifier **`protected`**. Using `protected` access offers an intermediate level of access between `public` and `private`. A superclass's `protected` members can be accessed by members of that superclass, by members of its subclasses and by members of other classes in the *same package*—`protected` members also have *package access*.

All `public` and `protected` superclass members retain their original access modifier when they become members of the subclass—`public` members of the superclass become `public` members of the subclass, and `protected` members of the superclass become `protected` members of the subclass. A superclass's `private` members are *not* accessible outside the class itself. Rather, they're *hidden* from its subclasses and can be accessed only through the `public` or `protected` methods inherited from the superclass.

Subclass methods can refer to `public` and `protected` members inherited from the superclass simply by using the member names. When a subclass method *overrides* an inherited superclass method, the *superclass* version of the method can be accessed from the *subclass* by preceding the superclass method name with keyword **`super`** and a dot (`.`) separator. We discuss accessing overridden members of the superclass in Section 9.4.

Software Engineering Observation 9.1

Methods of a subclass cannot directly access `private` members of their superclass. A subclass can change the state of `private` superclass instance variables only through non-`private` methods provided in the superclass and inherited by the subclass.

Software Engineering Observation 9.2

Declaring `private` instance variables helps you test, debug and correctly modify systems. If a subclass could access its superclass's `private` instance variables, classes that inherit from that subclass could access the instance variables as well. This would propagate access to what should be `private` instance variables, and the benefits of information hiding would be lost.

9.4 Relationship Between Superclasses and Subclasses

We now use an inheritance hierarchy containing types of *employees* in a company's payroll application to discuss the relationship between a superclass and its subclass. In this company, *commission employees* (who will be represented as objects of a superclass) are paid a percentage of their sales, while *base-salaried commission employees* (who will be represented as objects of a subclass) receive a base salary *plus* a percentage of their sales.

We divide our discussion of the relationship between these classes into five examples. The first declares class `CommissionEmployee`, which directly inherits from class `Object` and declares as `private` instance variables a first name, last name, social security number, commission rate and gross (i.e., total) sales amount.

The second example declares class `BasePlusCommissionEmployee`, which also directly inherits from class `Object` and declares as `private` instance variables a first name, last name, social security number, commission rate, gross sales amount *and* base salary. We create this class by *writing every line of code* the class requires—we'll soon see that it's much more efficient to create it by inheriting from class `CommissionEmployee`.

The third example declares a new `BasePlusCommissionEmployee` class that *extends* class `CommissionEmployee` (i.e., a `BasePlusCommissionEmployee` *is a* `CommissionEmployee` who also has a base salary). This *software reuse lets us write much less code* when developing the new subclass. In this example, class `BasePlusCommissionEmployee` attempts to access class `CommissionEmployee`'s `private` members—this results in compilation errors, because the subclass *cannot* access the superclass's `private` instance variables.

The fourth example shows that if `CommissionEmployee`'s instance variables are declared as `protected`, the `BasePlusCommissionEmployee` subclass *can* access that data directly. Both `BasePlusCommissionEmployee` classes contain identical functionality, but we show how the inherited version is easier to create and manage.

After we discuss the convenience of using `protected` instance variables, we create the fifth example, which sets the `CommissionEmployee` instance variables back to `private` to enforce good software engineering. Then we show how the `BasePlusCommissionEmployee` subclass can use `CommissionEmployee`'s `public` methods to manipulate (in a controlled manner) the `private` instance variables inherited from `CommissionEmployee`.

9.4.1 Creating and Using a `CommissionEmployee` Class

We begin by declaring class `CommissionEmployee` (Fig. 9.4). Line 4 begins the class declaration and indicates that class `CommissionEmployee` **`extends`** (i.e., *inherits from*) class **`Object`** (from package `java.lang`). This causes class `CommissionEmployee` to inherit the class `Object`'s methods—class `Object` does not have any fields. If you don't explicitly specify which class a new class extends, the class extends `Object` implicitly. For this reason, you typically will not include "`extends Object`" in your code—we do so in this one example only for demonstration purposes.

Overview of Class `CommissionEmployee`*'s Methods and Instance Variables*

Class `CommissionEmployee`'s `public` services include a constructor (lines 13–34) and methods `earnings` (lines 87–90) and `toString` (lines 93–101). Lines 37–52 declare `public` *get* methods for the class's `final` instance variables (declared in lines 6–8) `firstName`, `lastName` and `socialSecurityNumber`. These three instance variables are declared `final` because they do not need to be modified after they're initialized—this is also why we do

not provide corresponding *set* methods. Lines 55–84 declare `public` *set* and *get* methods for the class's `grossSales` and `commissionRate` instance variables (declared in lines 9–10). The class declares its instance variables as `private`, so objects of other classes cannot directly access these variables.

```
1  // Fig. 9.4: CommissionEmployee.java
2  // CommissionEmployee class represents an employee paid a
3  // percentage of gross sales.
4  public class CommissionEmployee extends Object
5  {
6     private final String firstName;
7     private final String lastName;
8     private final String socialSecurityNumber;
9     private double grossSales; // gross weekly sales
10    private double commissionRate; // commission percentage
11
12    // five-argument constructor
13    public CommissionEmployee(String firstName, String lastName,
14       String socialSecurityNumber, double grossSales,
15       double commissionRate)
16    {
17       // implicit call to Object's default constructor occurs here
18
19       // if grossSales is invalid throw exception
20       if (grossSales < 0.0)
21          throw new IllegalArgumentException(
22             "Gross sales must be >= 0.0");
23
24       // if commissionRate is invalid throw exception
25       if (commissionRate <= 0.0 || commissionRate >= 1.0)
26          throw new IllegalArgumentException(
27             "Commission rate must be > 0.0 and < 1.0");
28
29       this.firstName = firstName;
30       this.lastName = lastName;
31       this.socialSecurityNumber = socialSecurityNumber;
32       this.grossSales = grossSales;
33       this.commissionRate = commissionRate;
34    } // end constructor
35
36    // return first name
37    public String getFirstName()
38    {
39       return firstName;
40    }
41
42    // return last name
43    public String getLastName()
44    {
45       return lastName;
46    }
```

Fig. 9.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 1 of 3.)

```
47
48     // return social security number
49     public String getSocialSecurityNumber()
50     {
51        return socialSecurityNumber;
52     }
53
54     // set gross sales amount
55     public void setGrossSales(double grossSales)
56     {
57        if (grossSales < 0.0)
58           throw new IllegalArgumentException(
59              "Gross sales must be >= 0.0");
60
61        this.grossSales = grossSales;
62     }
63
64     // return gross sales amount
65     public double getGrossSales()
66     {
67        return grossSales;
68     }
69
70     // set commission rate
71     public void setCommissionRate(double commissionRate)
72     {
73        if (commissionRate <= 0.0 || commissionRate >= 1.0)
74           throw new IllegalArgumentException(
75              "Commission rate must be > 0.0 and < 1.0");
76
77        this.commissionRate = commissionRate;
78     }
79
80     // return commission rate
81     public double getCommissionRate()
82     {
83        return commissionRate;
84     }
85
86     // calculate earnings
87     public double earnings()
88     {
89        return commissionRate * grossSales;
90     }
91
92     // return String representation of CommissionEmployee object
93     @Override // indicates that this method overrides a superclass method
94     public String toString()
95     {
96        return String.format("%s: %s %s%n%s: %s%n%s: %.2f%n%s: %.2f",
97           "commission employee", firstName, lastName,
98           "social security number", socialSecurityNumber,
```

Fig. 9.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 2 of 3.)

```
 99            "gross sales", grossSales,
100            "commission rate", commissionRate);
101     }
102  } // end class CommissionEmployee
```

Fig. 9.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 3 of 3.)

Class `CommissionEmployee`'s Constructor

Constructors are *not* inherited, so class `CommissionEmployee` does not inherit class `Object`'s constructor. However, a superclass's constructors are still available to be called by subclasses. In fact, Java requires that *the first task of any subclass constructor is to call its direct superclass's constructor*, either explicitly or implicitly (if no constructor call is specified), to ensure that the instance variables inherited from the superclass are initialized properly. The syntax for calling a superclass constructor explicitly is discussed in Section 9.4.3. In this example, class `CommissionEmployee`'s constructor calls class `Object`'s constructor implicitly. If the code does not include an explicit call to the superclass constructor, Java *implicitly* calls the superclass's default or *no-argument* constructor. The comment in line 17 of Fig. 9.4 indicates where the implicit call to the superclass `Object`'s default constructor is made (you do *not* write the code for this call). `Object`'s default constructor does nothing. Even if a class does not have constructors, the default constructor that the compiler implicitly declares for the class will call the superclass's default or no-argument constructor.

After the implicit call to `Object`'s constructor, lines 20–22 and 25–27 validate the `grossSales` and `commissionRate` arguments. If these are valid (that is, the constructor does not throw an `IllegalArgumentException`), lines 29–33 assign the constructor's arguments to the class's instance variables.

We did not validate the values of arguments `firstName`, `lastName` and `socialSecurityNumber` before assigning them to the corresponding instance variables. We could validate the first and last names—perhaps to ensure that they're of a reasonable length. Similarly, a social security number could be validated using regular expressions (Section 14.7) to ensure that it contains nine digits, with or without dashes (e.g., `123-45-6789` or `123456789`).

Class `CommissionEmployee`'s `earnings` Method

Method `earnings` (lines 87–90) calculates a `CommissionEmployee`'s earnings. Line 89 multiplies the `commissionRate` by the `grossSales` and returns the result.

Class `CommissionEmployee`'s `toString` Method and the `@Override` Annotation

Method `toString` (lines 93–101) is special—it's one of the methods that *every* class inherits directly or indirectly from class `Object` (summarized in Section 9.6). Method `toString` returns a `String` representing an object. It's called implicitly whenever an object must be converted to a `String` representation, such as when an object is output by `printf` or output by `String` method `format` via the `%s` format specifier. Class `Object`'s `toString` method returns a `String` that includes the name of the object's class. It's primarily a placeholder that can be *overridden* by a subclass to specify an appropriate `String` representation of the data in a subclass object. Method `toString` of class `CommissionEmployee` overrides (redefines) class `Object`'s `toString` method. When invoked, `CommissionEmployee`'s `toString`

method uses `String` method `format` to return a `String` containing information about the `CommissionEmployee`. To override a superclass method, a subclass must declare a method with the *same signature* (method name, number of parameters, parameter types and order of parameter types) as the superclass method—`Object`'s `toString` method takes no parameters, so `CommissionEmployee` declares `toString` with no parameters.

Line 93 uses the optional **`@Override` annotation** to indicate that the following method declaration (i.e., `toString`) should *override* an *existing* superclass method. This annotation helps the compiler catch a few common errors. For example, in this case, you intend to override superclass method `toString`, which is spelled with a lowercase "`t`" and an uppercase "`S`." If you inadvertently use a lowercase "`s`," the compiler will flag this as an error because the superclass does not contain a method named `toString`. If you didn't use the `@Override` annotation, `toString` would be an entirely different method that would *not* be called if a `CommissionEmployee` were used where a `String` was needed.

Another common overriding error is declaring the wrong number or types of parameters in the parameter list. This creates an *unintentional overload* of the superclass method, rather than overriding the existing method. If you then attempt to call the method (with the correct number and types of parameters) on a subclass object, the superclass's version is invoked—potentially leading to subtle logic errors. When the compiler encounters a method declared with `@Override`, it compares the method's signature with the superclass's method signatures. If there isn't an exact match, the compiler issues an error message, such as "method does not override or implement a method from a supertype." You would then correct your method's signature so that it matches one in the superclass.

Error-Prevention Tip 9.1

Though the `@Override` annotation is optional, declare overridden methods with it to ensure at compilation time that you defined their signatures correctly. It's always better to find errors at compile time rather than at runtime. For this reason, the `toString` methods in Fig. 7.11 and in Chapter 8's examples should have been declared with `@Override`.

Common Programming Error 9.1

It's a compilation error to override a method with a more restricted access modifier—a `public` superclass method cannot become a `protected` or `private` subclass method; a `protected` superclass method cannot become a `private` subclass method. Doing so would break the is-a *relationship, which requires that all subclass objects be able to respond to method calls made to `public` methods declared in the superclass. If a `public` method, could be overridden as a `protected` or `private` method, the subclass objects would not be able to respond to the same method calls as superclass objects. Once a method is declared `public` in a superclass, the method remains `public` for all that class's direct and indirect subclasses.*

Class `CommissionEmployeeTest`

Figure 9.5 tests class `CommissionEmployee`. Lines 9–10 instantiate a `CommissionEmployee` object and invoke `CommissionEmployee`'s constructor (lines 13–34 of Fig. 9.4) to initialize it with `"Sue"` as the first name, `"Jones"` as the last name, `"222-22-2222"` as the social security number, `10000` as the gross sales amount ($10,000) and `.06` as the commission rate (i.e., 6%). Lines 15–24 use `CommissionEmployee`'s *get* methods to retrieve the object's instance-variable values for output. Lines 26–27 invoke the object's `setGrossSales` and `setCommissionRate` methods to change the values of instance variables `grossSales` and `commission-`

Rate. Lines 29–30 output the String representation of the updated CommissionEmployee. When an object is output using the %s format specifier, the object's toString method is invoked *implicitly* to obtain the object's String representation. [*Note:* In this chapter, we do not use the earnings method in each class, but it's used extensively in Chapter 10.]

```
1  // Fig. 9.5: CommissionEmployeeTest.java
2  // CommissionEmployee class test program.
3
4  public class CommissionEmployeeTest
5  {
6     public static void main(String[] args)
7     {
8        // instantiate CommissionEmployee object
9        CommissionEmployee employee = new CommissionEmployee(
10          "Sue", "Jones", "222-22-2222", 10000, .06);
11
12       // get commission employee data
13       System.out.println(
14          "Employee information obtained by get methods:");
15       System.out.printf("%n%s %s%n", "First name is",
16          employee.getFirstName());
17       System.out.printf("%s %s%n", "Last name is",
18          employee.getLastName());
19       System.out.printf("%s %s%n", "Social security number is",
20          employee.getSocialSecurityNumber());
21       System.out.printf("%s %.2f%n", "Gross sales is",
22          employee.getGrossSales());
23       System.out.printf("%s %.2f%n", "Commission rate is",
24          employee.getCommissionRate());
25
26       employee.setGrossSales(5000);
27       employee.setCommissionRate(.1);
28
29       System.out.printf("%n%s:%n%n%s%n",
30          "Updated employee information obtained by toString", employee);
31    } // end main
32 } // end class CommissionEmployeeTest
```

```
Employee information obtained by get methods:

First name is Sue
Last name is Jones
Social security number is 222-22-2222
Gross sales is 10000.00
Commission rate is 0.06

Updated employee information obtained by toString:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 5000.00
commission rate: 0.10
```

Fig. 9.5 | CommissionEmployee class test program.

9.4.2 Creating and Using a `BasePlusCommissionEmployee` Class

We now discuss the second part of our introduction to inheritance by declaring and testing (a completely new and independent) class `BasePlusCommissionEmployee` (Fig. 9.6), which contains a first name, last name, social security number, gross sales amount, commission rate *and* base salary. Class `BasePlusCommissionEmployee`'s `public` services include a `BasePlusCommissionEmployee` constructor (lines 15–42) and methods `earnings` (lines 111–114) and `toString` (lines 117–126). Lines 45–108 declare `public` *get* and *set* methods for the class's `private` instance variables (declared in lines 7–12) `firstName`, `lastName`, `socialSecurityNumber`, `grossSales`, `commissionRate` and `baseSalary`. These variables and methods encapsulate all the necessary features of a base-salaried commission employee. Note the *similarity* between this class and class `CommissionEmployee` (Fig. 9.4)—in this example, we'll not yet exploit that similarity.

```
1   // Fig. 9.6: BasePlusCommissionEmployee.java
2   // BasePlusCommissionEmployee class represents an employee who receives
3   // a base salary in addition to commission.
4
5   public class BasePlusCommissionEmployee
6   {
7      private final String firstName;
8      private final String lastName;
9      private final String socialSecurityNumber;
10     private double grossSales; // gross weekly sales
11     private double commissionRate; // commission percentage
12     private double baseSalary; // base salary per week
13
14     // six-argument constructor
15     public BasePlusCommissionEmployee(String firstName, String lastName,
16        String socialSecurityNumber, double grossSales,
17        double commissionRate, double baseSalary)
18     {
19        // implicit call to Object's default constructor occurs here
20
21        // if grossSales is invalid throw exception
22        if (grossSales < 0.0)
23           throw new IllegalArgumentException(
24              "Gross sales must be >= 0.0");
25
26        // if commissionRate is invalid throw exception
27        if (commissionRate <= 0.0 || commissionRate >= 1.0)
28           throw new IllegalArgumentException(
29              "Commission rate must be > 0.0 and < 1.0");
30
31        // if baseSalary is invalid throw exception
32        if (baseSalary < 0.0)
33           throw new IllegalArgumentException(
34              "Base salary must be >= 0.0");
35
```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 1 of 3.)

```
36          this.firstName = firstName;
37          this.lastName = lastName;
38          this.socialSecurityNumber = socialSecurityNumber;
39          this.grossSales = grossSales;
40          this.commissionRate = commissionRate;
41          this.baseSalary = baseSalary;
42       } // end constructor
43
44       // return first name
45       public String getFirstName()
46       {
47          return firstName;
48       }
49
50       // return last name
51       public String getLastName()
52       {
53          return lastName;
54       }
55
56       // return social security number
57       public String getSocialSecurityNumber()
58       {
59          return socialSecurityNumber;
60       }
61
62       // set gross sales amount
63       public void setGrossSales(double grossSales)
64       {
65          if (grossSales < 0.0)
66             throw new IllegalArgumentException(
67                "Gross sales must be >= 0.0");
68
69          this.grossSales = grossSales;
70       }
71
72       // return gross sales amount
73       public double getGrossSales()
74       {
75          return grossSales;
76       }
77
78       // set commission rate
79       public void setCommissionRate(double commissionRate)
80       {
81          if (commissionRate <= 0.0 || commissionRate >= 1.0)
82             throw new IllegalArgumentException(
83                "Commission rate must be > 0.0 and < 1.0");
84
85          this.commissionRate = commissionRate;
86       }
87
```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 2 of 3.)

```
88      // return commission rate
89      public double getCommissionRate()
90      {
91         return commissionRate;
92      }
93
94      // set base salary
95      public void setBaseSalary(double baseSalary)
96      {
97         if (baseSalary < 0.0)
98            throw new IllegalArgumentException(
99               "Base salary must be >= 0.0");
100
101        this.baseSalary = baseSalary;
102     }
103
104     // return base salary
105     public double getBaseSalary()
106     {
107        return baseSalary;
108     }
109
110     // calculate earnings
111     public double earnings()
112     {
113        return baseSalary + (commissionRate * grossSales);
114     }
115
116     // return String representation of BasePlusCommissionEmployee
117     @Override
118     public String toString()
119     {
120        return String.format(
121           "%s: %s %s%n%s: %s%n%s: %.2f%n%s: %.2f%n%s: %.2f",
122           "base-salaried commission employee", firstName, lastName,
123           "social security number", socialSecurityNumber,
124           "gross sales", grossSales, "commission rate", commissionRate,
125           "base salary", baseSalary);
126     }
127  } // end class BasePlusCommissionEmployee
```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 3 of 3.)

Class `BasePlusCommissionEmployee` does *not* specify "`extends Object`" in line 5, so the class *implicitly* extends `Object`. Also, like class `CommissionEmployee`'s constructor (lines 13–34 of Fig. 9.4), class `BasePlusCommissionEmployee`'s constructor invokes class `Object`'s default constructor *implicitly*, as noted in the comment in line 19.

Class `BasePlusCommissionEmployee`'s `earnings` method (lines 111–114) returns the result of adding the `BasePlusCommissionEmployee`'s base salary to the product of the commission rate and the employee's gross sales.

Class `BasePlusCommissionEmployee` overrides `Object` method `toString` to return a `String` containing the `BasePlusCommissionEmployee`'s information. Once again, we use

format specifier `%.2f` to format the gross sales, commission rate and base salary with two digits of precision to the right of the decimal point (line 121).

Testing Class **`BasePlusCommissionEmployee`**

Figure 9.7 tests class `BasePlusCommissionEmployee`. Lines 9–11 create a `BasePlusCommissionEmployee` object and pass `"Bob"`, `"Lewis"`, `"333-33-3333"`, `5000`, `.04` and `300` to the constructor as the first name, last name, social security number, gross sales, commission rate and base salary, respectively. Lines 16–27 use `BasePlusCommissionEmployee`'s *get* methods to retrieve the values of the object's instance variables for output. Line 29 invokes the object's `setBaseSalary` method to change the base salary. Method `setBaseSalary` (Fig. 9.6, lines 95–102) ensures that instance variable `baseSalary` is not assigned a negative value. Line 33 of Fig. 9.7 invokes method `toString` *explicitly* to get the object's `String` representation.

```
 1  // Fig. 9.7: BasePlusCommissionEmployeeTest.java
 2  // BasePlusCommissionEmployee test program.
 3
 4  public class BasePlusCommissionEmployeeTest
 5  {
 6     public static void main(String[] args)
 7     {
 8        // instantiate BasePlusCommissionEmployee object
 9        BasePlusCommissionEmployee employee =
10           new BasePlusCommissionEmployee(
11           "Bob", "Lewis", "333-33-3333", 5000, .04, 300);
12
13        // get base-salaried commission employee data
14        System.out.println(
15           "Employee information obtained by get methods:%n");
16        System.out.printf("%s %s%n", "First name is",
17           employee.getFirstName());
18        System.out.printf("%s %s%n", "Last name is",
19           employee.getLastName());
20        System.out.printf("%s %s%n", "Social security number is",
21           employee.getSocialSecurityNumber());
22        System.out.printf("%s %.2f%n", "Gross sales is",
23           employee.getGrossSales());
24        System.out.printf("%s %.2f%n", "Commission rate is",
25           employee.getCommissionRate());
26        System.out.printf("%s %.2f%n", "Base salary is",
27           employee.getBaseSalary());
28
29        employee.setBaseSalary(1000);
30
31        System.out.printf("%n%s:%n%n%s%n",
32           "Updated employee information obtained by toString",
33            employee.toString());
34     } // end main
35  } // end class BasePlusCommissionEmployeeTest
```

Fig. 9.7 | `BasePlusCommissionEmployee` test program. (Part 1 of 2.)

```
Employee information obtained by get methods:

First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales is 5000.00
Commission rate is 0.04
Base salary is 300.00

Updated employee information obtained by toString:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00
```

Fig. 9.7 | `BasePlusCommissionEmployee` test program. (Part 2 of 2.)

Notes on Class `BasePlusCommissionEmployee`

Much of class `BasePlusCommissionEmployee`'s code (Fig. 9.6) is *similar*, or *identical*, to that of class `CommissionEmployee` (Fig. 9.4). For example, `private` instance variables `firstName` and `lastName` and methods `setFirstName`, `getFirstName`, `setLastName` and `getLastName` are identical to those of class `CommissionEmployee`. The classes also both contain `private` instance variables `socialSecurityNumber`, `commissionRate` and `grossSales`, and corresponding *get* and *set* methods. In addition, the `BasePlusCommissionEmployee` constructor is *almost* identical to that of class `CommissionEmployee`, except that `BasePlusCommissionEmployee`'s constructor also sets the `baseSalary`. The other additions to class `BasePlusCommissionEmployee` are `private` instance variable `baseSalary` and methods `setBaseSalary` and `getBaseSalary`. Class `BasePlusCommissionEmployee`'s `toString` method is *almost* identical to that of class `CommissionEmployee` except that it also outputs instance variable `baseSalary` with two digits of precision to the right of the decimal point.

We literally *copied* code from class `CommissionEmployee` and *pasted* it into class `BasePlusCommissionEmployee`, then modified class `BasePlusCommissionEmployee` to include a base salary and methods that manipulate the base salary. This *"copy-and-paste" approach* is often error prone and time consuming. Worse yet, it spreads copies of the same code throughout a system, creating code-maintenance problems—changes to the code would need to be made in multiple classes. Is there a way to "acquire" the instance variables and methods of one class in a way that makes them part of other classes *without duplicating code*? Next we answer this question, using a more elegant approach to building classes that emphasizes the benefits of inheritance.

Software Engineering Observation 9.3

With inheritance, the instance variables and methods that are the same for all the classes in the hierarchy are declared in a superclass. Changes made to these common features in the superclass are inherited by the subclass. Without inheritance, changes would need to be made to all *the source-code files that contain a* copy *of the code in question.*

9.4.3 Creating a CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy

Now we declare class BasePlusCommissionEmployee (Fig. 9.8) to *extend* class CommissionEmployee (Fig. 9.4). A BasePlusCommissionEmployee object *is a* CommissionEmployee, because inheritance passes on class CommissionEmployee's capabilities. Class BasePlusCommissionEmployee also has instance variable baseSalary (Fig. 9.8, line 6). Keyword extends (line 4) indicates inheritance. BasePlusCommissionEmployee *inherits* CommissionEmployee's instance variables and methods.

Software Engineering Observation 9.4

At the design stage in an object-oriented system, you'll often find that certain classes are closely related. You should "factor out" common instance variables and methods and place them in a superclass. Then use inheritance to develop subclasses, specializing them with capabilities beyond those inherited from the superclass.

Software Engineering Observation 9.5

Declaring a subclass does not affect its superclass's source code. Inheritance preserves the integrity of the superclass.

Only CommissionEmployee's public and protected members are directly accessible in the subclass. The CommissionEmployee constructor is *not* inherited. So, the public BasePlusCommissionEmployee services include its constructor (lines 9–23), public methods inherited from CommissionEmployee, and methods setBaseSalary (lines 26–33), getBaseSalary (lines 36–39), earnings (lines 42–47) and toString (lines 50–60). Methods earnings and toString *override* the corresponding methods in class CommissionEmployee because their superclass versions do not properly calculate a BasePlusCommissionEmployee's earnings or return an appropriate String representation, respectively.

```
 1  // Fig. 9.8: BasePlusCommissionEmployee.java
 2  // private superclass members cannot be accessed in a subclass.
 3
 4  public class BasePlusCommissionEmployee extends CommissionEmployee
 5  {
 6     private double baseSalary; // base salary per week
 7
 8     // six-argument constructor
 9     public BasePlusCommissionEmployee(String firstName, String lastName,
10        String socialSecurityNumber, double grossSales,
11        double commissionRate, double baseSalary)
12     {
13        // explicit call to superclass CommissionEmployee constructor
14        super(firstName, lastName, socialSecurityNumber,
15           grossSales, commissionRate);
16
17        // if baseSalary is invalid throw exception
18        if (baseSalary < 0.0)
19           throw new IllegalArgumentException(
20              "Base salary must be >= 0.0");
21
```

Fig. 9.8 | private superclass members cannot be accessed in a subclass. (Part 1 of 3.)

```
22          this.baseSalary = baseSalary;
23       }
24
25       // set base salary
26       public void setBaseSalary(double baseSalary)
27       {
28          if (baseSalary < 0.0)
29             throw new IllegalArgumentException(
30                "Base salary must be >= 0.0");
31
32          this.baseSalary = baseSalary;
33       }
34
35       // return base salary
36       public double getBaseSalary()
37       {
38          return baseSalary;
39       }
40
41       // calculate earnings
42       @Override
43       public double earnings()
44       {
45          // not allowed: commissionRate and grossSales private in superclass
46          return baseSalary + (commissionRate * grossSales);
47       }
48
49       // return String representation of BasePlusCommissionEmployee
50       @Override
51       public String toString()
52       {
53          // not allowed: attempts to access private superclass members
54          return String.format(
55             "%s: %s %s%n%s: %s%n%s: %.2f%n%s: %.2f%n%s: %.2f",
56             "base-salaried commission employee", firstName, lastName,
57             "social security number", socialSecurityNumber,
58             "gross sales", grossSales, "commission rate", commissionRate,
59             "base salary", baseSalary);
60       }
61    } // end class BasePlusCommissionEmployee
```

```
BasePlusCommissionEmployee.java:46: error: commissionRate has private access
in CommissionEmployee
      return baseSalary + (commissionRate * grossSales);
                           ^
BasePlusCommissionEmployee.java:46: error: grossSales has private access in
CommissionEmployee
      return baseSalary + (commissionRate * grossSales);
                                            ^
BasePlusCommissionEmployee.java:56: error: firstName has private access in
CommissionEmployee
         "base-salaried commission employee", firstName, lastName,
                                              ^
```

Fig. 9.8 | `private` superclass members cannot be accessed in a subclass. (Part 2 of 3.)

```
BasePlusCommissionEmployee.java:56: error: lastName has private access in Com-
missionEmployee
         "base-salaried commission employee", firstName, lastName,
                                                          ^
BasePlusCommissionEmployee.java:57: error: socialSecurityNumber has private
access in CommissionEmployee
         "social security number", socialSecurityNumber,
                                   ^
BasePlusCommissionEmployee.java:58: error: grossSales has private access in
CommissionEmployee
         "gross sales", grossSales, "commission rate", commissionRate,
                        ^
BasePlusCommissionEmployee.java:58: error: commissionRate has private access
inCommissionEmployee
         "gross sales", grossSales, "commission rate", commissionRate,
                                                        ^
```

Fig. 9.8 | `private` superclass members cannot be accessed in a subclass. (Part 3 of 3.)

A Subclass's Constructor Must Call Its Superclass's Constructor

Each subclass constructor must implicitly or explicitly call one of its superclass's constructors to initialize the instance variables inherited from the superclass. Lines 14–15 in `BasePlusCommissionEmployee`'s six-argument constructor (lines 9–23) explicitly call class `CommissionEmployee`'s five-argument constructor (declared at lines 13–34 of Fig. 9.4) to initialize the superclass portion of a `BasePlusCommissionEmployee` object (i.e., variables `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate`). We do this by using the **superclass constructor call syntax**—keyword `super`, followed by a set of parentheses containing the superclass constructor arguments, which are used to initialize the superclass instance variables `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate`, respectively. If `BasePlusCommissionEmployee`'s constructor did not invoke the superclass's constructor explicitly, the compiler would attempt to insert a call to the superclass's default or no-argument constructor. Class `CommissionEmployee` does not have such a constructor, so the compiler would issue an error. The explicit superclass constructor call in lines 14–15 of Fig. 9.8 must be the *first* statement in the constructor's body. When a superclass contains a no-argument constructor, you can use `super()` to call that constructor explicitly, but this is rarely done.

Software Engineering Observation 9.6

You learned previously that you should not call a class's instance methods from its constructors and that we'll say why in Chapter 10. Calling a superclass constructor from a subclass constructor does not contradict this advice.

`BasePlusCommissionEmployee` *Methods* ***`Earnings`*** *and* ***`toString`***

The compiler generates errors for line 46 (Fig. 9.8) because `CommissionEmployee`'s instance variables `commissionRate` and `grossSales` are `private`—subclass `BasePlusCommissionEmployee`'s methods are *not* allowed to access superclass `CommissionEmployee`'s `private` instance variables. The compiler issues additional errors at lines 56–58 of `BasePlusCommissionEmployee`'s `toString` method for the same reason. The errors in `BasePlusCommissionEmployee` could have been prevented by using the *get* methods inherited from class `CommissionEmployee`. For example, line 46 could have called `getCommis-`

sionRate and getGrossSales to access CommissionEmployee's private instance variables commissionRate and grossSales, respectively. Lines 56–58 also could have used appropriate *get* methods to retrieve the values of the superclass's instance variables.

9.4.4 CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy Using protected Instance Variables

To enable class BasePlusCommissionEmployee to directly access superclass instance variables firstName, lastName, socialSecurityNumber, grossSales and commissionRate, we can declare those members as protected in the superclass. As we discussed in Section 9.3, a superclass's protected members are accessible by all subclasses of that superclass. In the new CommissionEmployee class, we modified only lines 6–10 of Fig. 9.4 to declare the instance variables with the protected access modifier as follows:

```
protected final String firstName;
protected final String lastName;
protected final String socialSecurityNumber;
protected double grossSales; // gross weekly sales
protected double commissionRate; // commission percentage
```

The rest of the class declaration (which is not shown here) is identical to that of Fig. 9.4.

We could have declared CommissionEmployee's instance variables public to enable subclass BasePlusCommissionEmployee to access them. However, declaring public instance variables is poor software engineering because it allows unrestricted access to the these variables from any class, greatly increasing the chance of errors. With protected instance variables, the subclass gets access to the instance variables, but classes that are not subclasses and classes that are not in the same package cannot access these variables directly—recall that protected class members are also visible to other classes in the same package.

Class **BasePlusCommissionEmployee**

Class BasePlusCommissionEmployee (Fig. 9.9) extends the new version of class CommissionEmployee with protected instance variables. BasePlusCommissionEmployee objects inherit CommissionEmployee's protected instance variables firstName, lastName, socialSecurityNumber, grossSales and commissionRate—all these variables are now protected members of BasePlusCommissionEmployee. As a result, the compiler does not generate errors when compiling line 45 of method earnings and lines 54–56 of method toString. If another class extends this version of class BasePlusCommissionEmployee, the new subclass also can access the protected members.

```
1   // Fig. 9.9: BasePlusCommissionEmployee.java
2   // BasePlusCommissionEmployee inherits protected instance
3   // variables from CommissionEmployee.
4
5   public class BasePlusCommissionEmployee extends CommissionEmployee
6   {
7      private double baseSalary; // base salary per week
```

Fig. 9.9 | BasePlusCommissionEmployee inherits protected instance variables from CommissionEmployee. (Part 1 of 2.)

```
 8
 9     // six-argument constructor
10     public BasePlusCommissionEmployee(String firstName, String lastName,
11        String socialSecurityNumber, double grossSales,
12        double commissionRate, double baseSalary)
13     {
14        super(firstName, lastName, socialSecurityNumber,
15           grossSales, commissionRate);
16
17        // if baseSalary is invalid throw exception
18        if (baseSalary < 0.0)
19           throw new IllegalArgumentException(
20              "Base salary must be >= 0.0");
21
22        this.baseSalary = baseSalary;
23     }
24
25     // set base salary
26     public void setBaseSalary(double baseSalary)
27     {
28        if (baseSalary < 0.0)
29           throw new IllegalArgumentException(
30              "Base salary must be >= 0.0");
31
32        this.baseSalary = baseSalary;
33     }
34
35     // return base salary
36     public double getBaseSalary()
37     {
38        return baseSalary;
39     }
40
41     // calculate earnings
42     @Override // indicates that this method overrides a superclass method
43     public double earnings()
44     {
45        return baseSalary + (commissionRate * grossSales);
46     }
47
48     // return String representation of BasePlusCommissionEmployee
49     @Override
50     public String toString()
51     {
52        return String.format(
53           "%s: %s %s%n%s: %s%n%s: %.2f%n%s: %.2f%n%s: %.2f",
54           "base-salaried commission employee", firstName, lastName,
55           "social security number", socialSecurityNumber,
56           "gross sales", grossSales, "commission rate", commissionRate,
57           "base salary", baseSalary);
58     }
59  } // end class BasePlusCommissionEmployee
```

Fig. 9.9 | `BasePlusCommissionEmployee` inherits `protected` instance variables from `CommissionEmployee`. (Part 2 of 2.)

A Subclass Object Contains the Instance Variables of All of Its Superclasses

When you create a `BasePlusCommissionEmployee` object, it contains all instance variables declared in the class hierarchy to that point—that is, those from classes `Object` (which does not have instance variables), `CommissionEmployee` and `BasePlusCommissionEmployee`. Class `BasePlusCommissionEmployee` does *not* inherit `CommissionEmployee`'s five-argument constructor, but *explicitly invokes* it (lines 14–15) to initialize the instance variables that `BasePlusCommissionEmployee` inherited from `CommissionEmployee`. Similarly, `CommissionEmployee`'s constructor *implicitly* calls class `Object`'s constructor. `BasePlusCommissionEmployee`'s constructor must *explicitly* call `CommissionEmployee`'s constructor because `CommissionEmployee` does *not* have a no-argument constructor that could be invoked implicitly.

Testing Class `BasePlusCommissionEmployee`

The `BasePlusCommissionEmployeeTest` class for this example is identical to that of Fig. 9.7 and produces the same output, so we do not show it here. Although the version of class `BasePlusCommissionEmployee` in Fig. 9.6 does not use inheritance and the version in Fig. 9.9 does, both classes provide the *same* functionality. The source code in Fig. 9.9 (59 lines) is considerably shorter than that in Fig. 9.6 (127 lines), because most of the class's functionality is now inherited from `CommissionEmployee`—there's now only one copy of the `CommissionEmployee` functionality. This makes the code easier to maintain, modify and debug, because the code related to a `CommissionEmployee` exists only in that class.

Notes on Using `protected` *Instance Variables*

In this example, we declared superclass instance variables as `protected` so that subclasses could access them. Inheriting `protected` instance variables enables direct access to the variables by subclasses. In most cases, however, it's better to use `private` instance variables to encourage proper software engineering. Your code will be easier to maintain, modify and debug.

Using `protected` instance variables creates several potential problems. First, the subclass object can set an inherited variable's value directly without using a *set* method. Therefore, a subclass object can assign an invalid value to the variable, possibly leaving the object in an inconsistent state. For example, if we were to declare `CommissionEmployee`'s instance variable `grossSales` as `protected`, a subclass object (e.g., `BasePlusCommissionEmployee`) could then assign a negative value to `grossSales`. Another problem with using `protected` instance variables is that subclass methods are more likely to be written so that they depend on the superclass's data implementation. In practice, subclasses should depend only on the superclass services (i.e., non-`private` methods) and not on the superclass data implementation. With `protected` instance variables in the superclass, we may need to modify all the subclasses of the superclass if the superclass implementation changes. For example, if for some reason we were to change the names of instance variables `firstName` and `lastName` to `first` and `last`, then we would have to do so for all occurrences in which a subclass directly references superclass instance variables `firstName` and `lastName`. Such a class is said to be **fragile** or **brittle**, because a small change in the superclass can "break" subclass implementation. You should be able to change the superclass implementation while still providing the same services to the subclasses. Of course, if the superclass services change, we must reimplement our subclasses. A third problem is that a class's `protected` members

are visible to all classes in the same package as the class containing the protected members—this is not always desirable.

Software Engineering Observation 9.7

Use the protected access modifier when a superclass should provide a method only to its subclasses and other classes in the same package, but not to other clients.

Software Engineering Observation 9.8

Declaring superclass instance variables private (as opposed to protected) enables the superclass implementation of these instance variables to change without affecting subclass implementations.

Error-Prevention Tip 9.2

When possible, do not include protected instance variables in a superclass. Instead, include non-private methods that access private instance variables. This will help ensure that objects of the class maintain consistent states.

9.4.5 CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy Using private Instance Variables

Let's reexamine our hierarchy once more, this time using good software engineering practices.

Class CommissionEmployee

Class CommissionEmployee (Fig. 9.10) declares instance variables firstName, lastName, socialSecurityNumber, grossSales and commissionRate as *private* (lines 6–10) and provides public methods getFirstName, getLastName, getSocialSecurityNumber, setGrossSales, getGrossSales, setCommissionRate, getCommissionRate, earnings and toString for manipulating these values. Methods earnings (lines 87–90) and toString (lines 93–101) use the class's *get* methods to obtain the values of its instance variables. If we decide to change the names of the instance variables, the earnings and toString declarations will *not* require modification—only the bodies of the *get* and *set* methods that directly manipulate the instance variables will need to change. These changes occur solely within the superclass—no changes to the subclass are needed. *Localizing the effects of changes* like this is a good software engineering practice.

```
1  // Fig. 9.10: CommissionEmployee.java
2  // CommissionEmployee class uses methods to manipulate its
3  // private instance variables.
4  public class CommissionEmployee
5  {
6     private final String firstName;
7     private final String lastName;
8     private final String socialSecurityNumber;
9     private double grossSales; // gross weekly sales
10    private double commissionRate; // commission percentage
```

Fig. 9.10 | CommissionEmployee class uses methods to manipulate its private instance variables. (Part 1 of 3.)

```
11
12      // five-argument constructor
13      public CommissionEmployee(String firstName, String lastName,
14         String socialSecurityNumber, double grossSales,
15         double commissionRate)
16      {
17         // implicit call to Object constructor occurs here
18
19         // if grossSales is invalid throw exception
20         if (grossSales < 0.0)
21            throw new IllegalArgumentException(
22               "Gross sales must be >= 0.0");
23
24         // if commissionRate is invalid throw exception
25         if (commissionRate <= 0.0 || commissionRate >= 1.0)
26            throw new IllegalArgumentException(
27               "Commission rate must be > 0.0 and < 1.0");
28
29         this.firstName = firstName;
30         this.lastName = lastName;
31         this.socialSecurityNumber = socialSecurityNumber;
32         this.grossSales = grossSales;
33         this.commissionRate = commissionRate;
34      } // end constructor
35
36      // return first name
37      public String getFirstName()
38      {
39         return firstName;
40      }
41
42      // return last name
43      public String getLastName()
44      {
45         return lastName;
46      }
47
48      // return social security number
49      public String getSocialSecurityNumber()
50      {
51         return socialSecurityNumber;
52      }
53
54      // set gross sales amount
55      public void setGrossSales(double grossSales)
56      {
57         if (grossSales < 0.0)
58            throw new IllegalArgumentException(
59               "Gross sales must be >= 0.0");
60
61         this.grossSales = grossSales;
62      }
```

Fig. 9.10 | `CommissionEmployee` class uses methods to manipulate its `private` instance variables. (Part 2 of 3.)

```
63
64      // return gross sales amount
65      public double getGrossSales()
66      {
67         return grossSales;
68      }
69
70      // set commission rate
71      public void setCommissionRate(double commissionRate)
72      {
73         if (commissionRate <= 0.0 || commissionRate >= 1.0)
74            throw new IllegalArgumentException(
75               "Commission rate must be > 0.0 and < 1.0");
76
77         this.commissionRate = commissionRate;
78      }
79
80      // return commission rate
81      public double getCommissionRate()
82      {
83         return commissionRate;
84      }
85
86      // calculate earnings
87      public double earnings()
88      {
89         return getCommissionRate() * getGrossSales();
90      }
91
92      // return String representation of CommissionEmployee object
93      @Override
94      public String toString()
95      {
96         return String.format("%s: %s %s%n%s: %s%n%s: %.2f%n%s: %.2f",
97            "commission employee", getFirstName(), getLastName(),
98            "social security number", getSocialSecurityNumber(),
99            "gross sales", getGrossSales(),
100           "commission rate", getCommissionRate());
101     }
102  } // end class CommissionEmployee
```

Fig. 9.10 | `CommissionEmployee` class uses methods to manipulate its `private` instance variables. (Part 3 of 3.)

Class `BasePlusCommissionEmployee`

Subclass `BasePlusCommissionEmployee` (Fig. 9.11) inherits `CommissionEmployee`'s non-`private` methods and can access (in a controlled way) the `private` superclass members via those methods. Class `BasePlusCommissionEmployee` has several changes that distinguish it from Fig. 9.9. Methods `earnings` (lines 43–47) and `toString` (lines 50–55) each invoke method `getBaseSalary` to obtain the base salary value, rather than accessing `baseSalary` directly. If we decide to rename instance variable `baseSalary`, only the bodies of method `setBaseSalary` and `getBaseSalary` will need to change.

```
1  // Fig. 9.11: BasePlusCommissionEmployee.java
2  // BasePlusCommissionEmployee class inherits from CommissionEmployee
3  // and accesses the superclass's private data via inherited
4  // public methods.
5
6  public class BasePlusCommissionEmployee extends CommissionEmployee
7  {
8     private double baseSalary; // base salary per week
9
10    // six-argument constructor
11    public BasePlusCommissionEmployee(String firstName, String lastName,
12       String socialSecurityNumber, double grossSales,
13       double commissionRate, double baseSalary)
14    {
15       super(firstName, lastName, socialSecurityNumber,
16          grossSales, commissionRate);
17
18       // if baseSalary is invalid throw exception
19       if (baseSalary < 0.0)
20          throw new IllegalArgumentException(
21             "Base salary must be >= 0.0");
22
23       this.baseSalary = baseSalary;
24    }
25
26    // set base salary
27    public void setBaseSalary(double baseSalary)
28    {
29       if (baseSalary < 0.0)
30          throw new IllegalArgumentException(
31             "Base salary must be >= 0.0");
32
33       this.baseSalary = baseSalary;
34    }
35
36    // return base salary
37    public double getBaseSalary()
38    {
39       return baseSalary;
40    }
41
42    // calculate earnings
43    @Override
44    public double earnings()
45    {
46       return getBaseSalary() + super.earnings();
47    }
48
49    // return String representation of BasePlusCommissionEmployee
50    @Override
51    public String toString()
52    {
```

Fig. 9.11 | `BasePlusCommissionEmployee` class inherits from `CommissionEmployee` and accesses the superclass's `private` data via inherited `public` methods. (Part 1 of 2.)

```
53          return String.format("%s %s%n%s: %.2f", "base-salaried",
54             super.toString(), "base salary", getBaseSalary());
55       }
56    } // end class BasePlusCommissionEmployee
```

Fig. 9.11 | `BasePlusCommissionEmployee` class inherits from `CommissionEmployee` and accesses the superclass's `private` data via inherited `public` methods. (Part 2 of 2.)

Class `BasePlusCommissionEmployee`*'s* `earnings` *Method*

Method `earnings` (lines 43–47) overrides class `CommissionEmployee`'s `earnings` method (Fig. 9.10, lines 87–90) to calculate a base-salaried commission employee's earnings. The new version obtains the portion of the earnings based on commission alone by calling `CommissionEmployee`'s `earnings` method with `super.earnings()` (line 46), then adds the base salary to this value to calculate the total earnings. Note the syntax used to invoke an *overridden* superclass method from a subclass—place the keyword `super` and a dot (`.`) separator before the superclass method name. This method invocation is a good software engineering practice—if a method performs all or some of the actions needed by another method, call that method rather than duplicate its code. By having `BasePlusCommissionEmployee`'s `earnings` method invoke `CommissionEmployee`'s `earnings` method to calculate part of a `BasePlusCommissionEmployee` object's earnings, we *avoid duplicating the code* and *reduce code-maintenance problems*.

Common Programming Error 9.2

When a superclass method is overridden in a subclass, the subclass version often calls the superclass version to do a portion of the work. Failure to prefix the superclass method name with the keyword `super` *and the dot (*`.`*) separator when calling the superclass's method causes the subclass method to call itself, potentially creating an error called infinite recursion, which would eventually cause the method-call stack to overflow—a fatal runtime error. Recursion, used correctly, is a powerful capability discussed in Chapter 18.*

Class `BasePlusCommissionEmployee`*'s* `toString` *Method*

Similarly, `BasePlusCommissionEmployee`'s `toString` method (Fig. 9.11, lines 50–55) overrides `CommissionEmployee`'s `toString` method (Fig. 9.10, lines 93–101) to return a `String` representation that's appropriate for a base-salaried commission employee. The new version creates part of a `BasePlusCommissionEmployee` object's `String` representation (i.e., the `String` `"commission employee"` and the values of class `CommissionEmployee`'s `private` instance variables) by calling `CommissionEmployee`'s `toString` method with the expression `super.toString()` (Fig. 9.11, line 54). `BasePlusCommissionEmployee`'s `toString` method then completes the remainder of a `BasePlusCommissionEmployee` object's `String` representation (i.e., the value of class `BasePlusCommissionEmployee`'s base salary).

Testing Class `BasePlusCommissionEmployee`

Class `BasePlusCommissionEmployeeTest` performs the same manipulations on a `BasePlusCommissionEmployee` object as in Fig. 9.7 and produces the same output, so we do not show it here. Although each `BasePlusCommissionEmployee` class you've seen behaves identically, the version in Fig. 9.11 is the best engineered. By using inheritance and by calling methods that hide the data and ensure consistency, we've efficiently and effectively constructed a well-engineered class.

9.5 Constructors in Subclasses

As we explained, instantiating a subclass object begins a chain of constructor calls in which the subclass constructor, before performing its own tasks, explicitly uses `super` to call one of the constructors in its direct superclass or implicitly calls the superclass's default or no-argument constructor. Similarly, if the superclass is derived from another class—true of every class except `Object`—the superclass constructor invokes the constructor of the next class up the hierarchy, and so on. The last constructor called in the chain is *always* `Object`'s constructor. The original subclass constructor's body finishes executing *last*. Each superclass's constructor manipulates the superclass instance variables that the subclass object inherits. For example, consider again the `CommissionEmployee`–`BasePlusCommissionEmployee` hierarchy from Figs. 9.10–9.11. When an app creates a `BasePlusCommissionEmployee` object, its constructor is called. That constructor calls `CommissionEmployee`'s constructor, which in turn calls `Object`'s constructor. Class `Object`'s constructor has an *empty body*, so it immediately returns control to `CommissionEmployee`'s constructor, which then initializes the `CommissionEmployee` instance variables that are part of the `BasePlusCommissionEmployee` object. When `CommissionEmployee`'s constructor completes execution, it returns control to `BasePlusCommissionEmployee`'s constructor, which initializes the `baseSalary`.

Software Engineering Observation 9.9

Java ensures that even if a constructor does not assign a value to an instance variable, the variable is still initialized to its default value (e.g., `0` *for primitive numeric types,* `false` *for* `boolean`*s,* `null` *for references).*

9.6 Class `Object`

As we discussed earlier in this chapter, all classes in Java inherit directly or indirectly from class `Object` (package `java.lang`), so its 11 methods (some are overloaded) are inherited by all other classes. Figure 9.12 summarizes `Object`'s methods. We discuss several `Object` methods throughout this book (as indicated in Fig. 9.12).

Method	Description
`equals`	This method compares two objects for equality and returns `true` if they're equal and `false` otherwise. The method takes any `Object` as an argument. When objects of a particular class must be compared for equality, the class should override method `equals` to compare the *contents* of the two objects. For the requirements of implementing this method (which include also overriding method `hashCode`), refer to the method's documentation at `docs.oracle.com/javase/7/docs/api/java/lang/Object.html#equals(java.lang.Object)`. The default `equals` implementation uses operator == to determine whether two references *refer to the same object* in memory. Section 14.3.3 demonstrates class `String`'s `equals` method and differentiates between comparing `String` objects with == and with `equals`.
`hashCode`	Hashcodes are `int` values used for high-speed storage and retrieval of information stored in a data structure that's known as a hashtable (see Section 16.11). This method is also called as part of `Object`'s default `toString` method implementation.

Fig. 9.12 | `Object` methods. (Part 1 of 2.)

Method	Description
`toString`	This method (introduced in Section 9.4.1) returns a `String` representation of an object. The default implementation of this method returns the package name and class name of the object's class typically followed by a hexadecimal representation of the value returned by the object's `hashCode` method.
`wait`, `notify`, `notifyAll`	Methods `notify`, `notifyAll` and the three overloaded versions of `wait` are related to multithreading, which is discussed in Chapter 23.
`getClass`	Every object in Java knows its own type at execution time. Method `getClass` (used in Sections 10.5– and 12.5) returns an object of class `Class` (package `java.lang`) that contains information about the object's type, such as its class name (returned by `Class` method `getName`).
`finalize`	This `protected` method is called by the garbage collector to perform termination housekeeping on an object just before the garbage collector reclaims the object's memory. Recall from Section 8.10 that it's unclear whether, or when, `finalize` will be called. For this reason, most programmers should avoid method `finalize`.
`clone`	This `protected` method, which takes no arguments and returns an `Object` reference, makes a copy of the object on which it's called. The default implementation performs a so-called **shallow copy**—instance-variable values in one object are copied into another object of the same type. For reference types, only the references are copied. A typical overridden `clone` method's implementation would perform a **deep copy** that creates a new object for each reference-type instance variable. *Implementing `clone` correctly is difficult. For this reason, its use is discouraged.* Some industry experts suggest that object serialization should be used instead. We discuss object serialization in Chapter 15. Recall from Chapter 7 that arrays are objects. As a result, like all other objects, arrays inherit the members of class `Object`. Every array has an overridden `clone` method that copies the array. However, if the array stores references to objects, the objects are not copied—a shallow copy is performed.

Fig. 9.12 | `Object` methods. (Part 2 of 2.)

9.7 (Optional) GUI and Graphics Case Study: Displaying Text and Images Using Labels

Programs often use labels when they need to display information or instructions to the user in a graphical user interface. **Labels** are a convenient way of identifying GUI components on the screen and keeping the user informed about the current state of the program. In Java, an object of class **`JLabel`** (from package `javax.swing`) can display text, an image or both. The example in Fig. 9.13 demonstrates several `JLabel` features, including a plain text label, an image label and a label with both text and an image.

```
1   // Fig 9.13: LabelDemo.java
2   // Demonstrates the use of labels.
3   import java.awt.BorderLayout;
4   import javax.swing.ImageIcon;
```

Fig. 9.13 | `JLabel` with text and with images. (Part 1 of 2.)

```
 5  import javax.swing.JLabel;
 6  import javax.swing.JFrame;
 7
 8  public class LabelDemo
 9  {
10     public static void main(String[] args)
11     {
12        // Create a label with plain text
13        JLabel northLabel = new JLabel("North");
14
15        // create an icon from an image so we can put it on a JLabel
16        ImageIcon labelIcon = new ImageIcon("GUItip.gif");
17
18        // create a label with an Icon instead of text
19        JLabel centerLabel = new JLabel(labelIcon);
20
21        // create another label with an Icon
22        JLabel southLabel = new JLabel(labelIcon);
23
24        // set the label to display text (as well as an icon)
25        southLabel.setText("South");
26
27        // create a frame to hold the labels
28        JFrame application = new JFrame();
29
30        application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
31
32        // add the labels to the frame; the second argument specifies
33        // where on the frame to add the label
34        application.add(northLabel, BorderLayout.NORTH);
35        application.add(centerLabel, BorderLayout.CENTER);
36        application.add(southLabel, BorderLayout.SOUTH);
37
38        application.setSize(300, 300);
39        application.setVisible(true);
40     } // end main
41  } // end class LabelDemo
```

Fig. 9.13 | `JLabel` with text and with images. (Part 2 of 2.)

Lines 3–6 import the classes we need to display `JLabel`s. `BorderLayout` from package `java.awt` contains constants that specify where we can place GUI components in the

JFrame. Class **ImageIcon** represents an image that can be displayed on a JLabel, and class JFrame represents the window that will contain all the labels.

Line 13 creates a JLabel that displays its constructor argument—the string "North". Line 16 declares local variable labelIcon and assigns it a new ImageIcon. The constructor for ImageIcon receives a String that specifies the path to the image. Since we specify only a filename, Java assumes that it's in the same directory as class LabelDemo. ImageIcon can load images in GIF, JPEG and PNG image formats. Line 19 declares and initializes local variable centerLabel with a JLabel that displays the labelIcon. Line 22 declares and initializes local variable southLabel with a JLabel similar to the one in line 19. However, line 25 calls method **setText** to change the text the label displays. Method setText can be called on any JLabel to change its text. This JLabel displays both the icon and the text.

Line 28 creates the JFrame that displays the JLabels, and line 30 indicates that the program should terminate when the JFrame is closed. We attach the labels to the JFrame in lines 34–36 by calling an overloaded version of method add that takes two parameters. The first parameter is the component we want to attach, and the second is the region in which it should be placed. Each JFrame has an associated **layout** that helps the JFrame position the GUI components that are attached to it. The default layout for a JFrame is known as a **BorderLayout** and has five regions—NORTH (top), SOUTH (bottom), EAST (right side), WEST (left side) and CENTER. Each of these is declared as a constant in class BorderLayout. When calling method add with one argument, the JFrame places the component in the CENTER automatically. If a position already contains a component, then the new component takes its place. Lines 38 and 39 set the size of the JFrame and make it visible on screen.

GUI and Graphics Case Study Exercise

9.1 Modify GUI and Graphics Case Study Exercise 8.1 to include a JLabel as a status bar that displays counts representing the number of each shape displayed. Class DrawPanel should declare a method that returns a String containing the status text. In main, first create the DrawPanel, then create the JLabel with the status text as an argument to the JLabel's constructor. Attach the JLabel to the SOUTH region of the JFrame, as shown in Fig. 9.14.

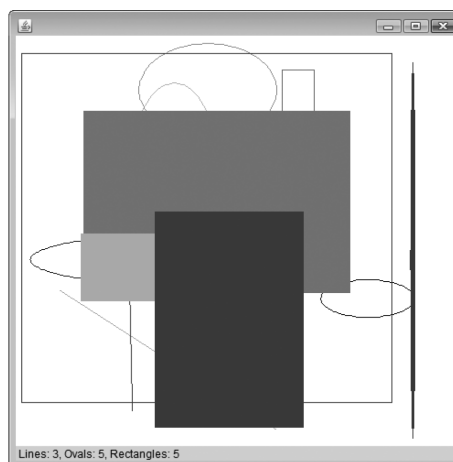

Fig. 9.14 | JLabel displaying shape statistics.

9.8 Wrap-Up

This chapter introduced inheritance—the ability to create classes by acquiring an existing class's members (without copying and pasting the code) and having the ability to embellish them with new capabilities. You learned the notions of superclasses and subclasses and used keyword `extends` to create a subclass that inherits members from a superclass. We showed how to use the `@Override` annotation to prevent unintended overloading by indicating that a method overrides a superclass method. We introduced the access modifier `protected`; subclass methods can directly access `protected` superclass members. You learned how to use `super` to access overridden superclass members. You also saw how constructors are used in inheritance hierarchies. Finally, you learned about the methods of class `Object`, the direct or indirect superclass of all Java classes.

In Chapter 10, Object-Oriented Programming: Polymorphism and Interfaces, we build on our discussion of inheritance by introducing *polymorphism*—an object-oriented concept that enables us to write programs that conveniently handle, in a more general and convenient manner, objects of a wide variety of classes related by inheritance. After studying Chapter 10, you'll be familiar with classes, objects, encapsulation, inheritance and polymorphism—the key technologies of object-oriented programming.

Summary

Section 9.1 Introduction

- Inheritance (p. 353) reduces program-development time.
- The direct superclass (p. 353) of a subclass is the one from which the subclass inherits. An indirect superclass (p. 353) of a subclass is two or more levels up the class hierarchy from that subclass.
- In single inheritance (p. 353), a class is derived from one superclass. In multiple inheritance, a class is derived from more than one direct superclass. Java does not support multiple inheritance.
- A subclass is more specific than its superclass and represents a smaller group of objects (p. 353).
- Every object of a subclass is also an object of that class's superclass. However, a superclass object is not an object of its class's subclasses.
- An *is-a* relationship (p. 354) represents inheritance. In an *is-a* relationship, an object of a subclass also can be treated as an object of its superclass.
- A *has-a* relationship (p. 354) represents composition. In a *has-a* relationship, a class object contains references to objects of other classes.

Section 9.2 Superclasses and Subclasses

- Single-inheritance relationships form treelike hierarchical structures—a superclass exists in a hierarchical relationship with its subclasses.

Section 9.3 `protected` Members

- A superclass's `public` members are accessible wherever the program has a reference to an object of that superclass or one of its subclasses.
- A superclass's `private` members can be accessed directly only within the superclass's declaration.
- A superclass's `protected` members (p. 356) have an intermediate level of protection between `public` and `private` access. They can be accessed by members of the superclass, by members of its subclasses and by members of other classes in the same package.

- A superclass's `private` members are hidden in its subclasses and can be accessed only through the `public` or `protected` methods inherited from the superclass.
- An overridden superclass method can be accessed from a subclass if the superclass method name is preceded by `super` (p. 356) and a dot (`.`) separator.

Section 9.4 Relationship Between Superclasses and Subclasses

- A subclass cannot access the `private` members of its superclass, but it can access the non-`private` members.
- A subclass can invoke a constructor of its superclass by using the keyword `super`, followed by a set of parentheses containing the superclass constructor arguments. This must appear as the first statement in the subclass constructor's body.
- A superclass method can be overridden in a subclass to declare an appropriate implementation for the subclass.
- The `@Override` annotation (p. 361) indicates that a method should override a superclass method. When the compiler encounters a method declared with `@Override`, it compares the method's signature with the superclass's method signatures. If there isn't an exact match, the compiler issues an error message, such as "method does not override or implement a method from a supertype."
- Method `toString` takes no arguments and returns a `String`. The `Object` class's `toString` method is normally overridden by a subclass.
- When an object is output using the `%s` format specifier, the object's `toString` method is called implicitly to obtain its `String` representation.

Section 9.5 Constructors in Subclasses

- The first task of a subclass constructor is to call its direct superclass's constructor (p. 370) to ensure that the instance variables inherited from the superclass are initialized.

Section 9.6 Class `Object`

- See the table of class `Object`'s methods in Fig. 9.12.

Self-Review Exercises

9.1 Fill in the blanks in each of the following statements:

a) ________ is a form of software reusability in which new classes acquire the members of existing classes and embellish those classes with new capabilities.
b) A superclass's ________ members can be accessed in the superclass declaration *and in* subclass declarations.
c) In a(n) ________ relationship, an object of a subclass can also be treated as an object of its superclass.
d) In a(n) ________ relationship, a class object has references to objects of other classes as members.
e) In single inheritance, a class exists in a(n) ________ relationship with its subclasses.
f) A superclass's ________ members are accessible anywhere that the program has a reference to an object of that superclass or to an object of one of its subclasses.
g) When an object of a subclass is instantiated, a superclass ________ is called implicitly or explicitly.
h) Subclass constructors can call superclass constructors via the ________ keyword.

9.2 State whether each of the following is *true* or *false*. If a statement is *false*, explain why.

a) Superclass constructors are not inherited by subclasses.
b) A *has-a* relationship is implemented via inheritance.

c) A `Car` class has an *is-a* relationship with the `SteeringWheel` and `Brakes` classes.
d) When a subclass redefines a superclass method by using the same signature, the subclass is said to overload that superclass method.

Answers to Self-Review Exercises

9.1 a) Inheritance. b) `public` and `protected`. c) *is-a* or inheritance. d) *has-a* or composition. e) hierarchical. f) `public`. g) constructor. h) `super`.

9.2 a) True. b) False. A *has-a* relationship is implemented via composition. An *is-a* relationship is implemented via inheritance. c) False. This is an example of a *has-a* relationship. Class `Car` has an *is-a* relationship with class `Vehicle`. d) False. This is known as overriding, not overloading—an overloaded method has the same name, but a different signature.

Exercises

9.3 *(Using Composition Rather Than Inheritance)* Many programs written with inheritance could be written with composition instead, and vice versa. Rewrite class `BasePlusCommissionEmployee` (Fig. 9.11) of the `CommissionEmployee–BasePlusCommissionEmployee` hierarchy to use composition rather than inheritance.

9.4 *(Software Reuse)* Discuss the ways in which inheritance promotes software reuse, saves time during program development and helps prevent errors.

9.5 *(`Student` Inheritance Hierarchy)* 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.]

9.6 *(`Shape` Inheritance Hierarchy)* The world of shapes is much richer than the shapes included in the inheritance hierarchy of Fig. 9.3. Write down all the shapes you can think of—both two-dimensional and three-dimensional—and form them into a more complete `Shape` hierarchy with as many levels as possible. Your hierarchy should have class `Shape` at the top. Classes `TwoDimensionalShape` and `ThreeDimensionalShape` should extend `Shape`. Add additional subclasses, such as `Quadrilateral` and `Sphere`, at their correct locations in the hierarchy as necessary.

9.7 *(`protected` vs. `private`)* 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.

9.8 *(`Quadrilateral` Inheritance Hierarchy)* Write an inheritance hierarchy for classes `Quadrilateral`, `Trapezoid`, `Parallelogram`, `Rectangle` and `Square`. Use `Quadrilateral` as the superclass of the hierarchy. Create and use a `Point` class to represent the points in each shape. Make the hierarchy as deep (i.e., as many levels) as possible. Specify the instance variables and methods for each class. The `private` instance variables of `Quadrilateral` should be the *x-y* coordinate pairs for the four endpoints of the `Quadrilateral`. Write a program that instantiates objects of your classes and outputs each object's area (except `Quadrilateral`).

9.9 *(What Does Each Code Snippet Do?)*
a) Assume that the following method call is located in an overridden `earnings` method in a subclass:

```
super.earnings()
```

b) Assume that the following line of code appears before a method declaration:

```
@Override
```

c) Assume that the following line of code appears as the first statement in a constructor's body:

```
super(firstArgument, secondArgument);
```

9.10 *(Write a Line of Code)* Write a line of code that performs each of the following tasks:

a) Specify that class `PieceWorker` inherits from class `Employee`.
b) Call superclass `Employee`'s `toString` method from subclass `PieceWorker`'s `toString` method.
c) Call superclass `Employee`'s constructor from subclass `PieceWorker`'s constructor—assume that the superclass constructor receives three `String`s representing the first name, last name and social security number.

9.11 *(Using **`super`** in a Constructor's Body)* Explain why you would use `super` in the first statement of a subclass constructor's body.

9.12 *(Using **`super`** in an Instance Method's Body)* Explain why you would use `super` in the body of a subclass's instance method.

9.13 *(Calling* get *Methods in a Class's Body)* In Figs. 9.10–9.11 methods `earnings` and `toString` each call various *get* methods within the same class. Explain the benefits of calling these *get* methods within the classes.

9.14 *(**`Employee`** Hierarchy)* In this chapter, you studied an inheritance hierarchy in which class `BasePlusCommissionEmployee` inherited from class `CommissionEmployee`. However, not all types of employees are `CommissionEmployee`s. In this exercise, you'll create a more general `Employee` superclass that *factors out* the attributes and behaviors in class `CommissionEmployee` that are common to all `Employee`s. The common attributes and behaviors for all `Employee`s are `firstName`, `lastName`, `socialSecurityNumber`, `getFirstName`, `getLastName`, `getSocialSecurityNumber` and a portion of method `toString`. Create a new superclass `Employee` that contains these instance variables and methods and a constructor. Next, rewrite class `CommissionEmployee` from Section 9.4.5 as a subclass of `Employee`. Class `CommissionEmployee` should contain only the instance variables and methods that are not declared in superclass `Employee`. Class `CommissionEmployee`'s constructor should invoke class `Employee`'s constructor and `CommissionEmployee`'s `toString` method should invoke `Employee`'s `toString` method. Once you've completed these modifications, run the `CommissionEmployeeTest` and `BasePlusCommissionEmployeeTest` apps using these new classes to ensure that the apps still display the same results for a `CommissionEmployee` object and `BasePlusCommissionEmployee` object, respectively.

9.15 *(Creating a New Subclass of **`Employee`**)* Other types of `Employee`s might include `SalariedEmployee`s who get paid a fixed weekly salary, `PieceWorker`s who get paid by the number of pieces they produce or `HourlyEmployee`s who get paid an hourly wage with time-and-a-half—1.5 times the hourly wage—for hours worked over 40 hours.

Create class `HourlyEmployee` that inherits from class `Employee` (Exercise 9.14) and has instance variable `hours` (a `double`) that represents the hours worked, instance variable `wage` (a `double`) that represents the wages per hour, a constructor that takes as arguments a first name, a last name, a social security number, an hourly wage and the number of hours worked, *set* and *get* methods for manipulating the `hours` and `wage`, an `earnings` method to calculate an `HourlyEmployee`'s earnings based on the hours worked and a `toString` method that returns the `HourlyEmployee`'s `String` representation. Method `setWage` should ensure that `wage` is nonnegative, and `setHours` should ensure that the value of hours is between 0 and 168 (the total number of hours in a week). Use class `HourlyEmployee` in a test program that's similar to the one in Fig. 9.5.