

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.

```
1 // Fig. 9.4: CommissionEmployee.java
2 // CommissionEmployee class represents an employee
3 // percentage of gross sales.
4 public class CommissionEmployee extends Object {
5     private final String firstName;
6     private final String lastName;
7     private final String socialSecurityNumber;
8     private double grossSales; // gross weekly sal
9     private double commissionRate; // commission p
10
11     // five-argument constructor
12     public CommissionEmployee(String firstName, St
13         String socialSecurityNumber, double grossSa
14         double commissionRate) {
15         // implicit call to Object's default constr
16
```

```

17         // if grossSales is invalid throw exception
18         if (grossSales < 0.0) {
19             throw new IllegalArgumentException("Gros
20                 }
21
22         // if commissionRate is invalid throw excep
23         if (commissionRate <= 0.0 || commissionRate
24             throw new IllegalArgumentException(
25                 "Commission rate must be > 0.0 and <
26                 }
27
28         this.firstName = firstName;
29         this.lastName = lastName;
30         this.socialSecurityNumber = socialSecurityN
31         this.grossSales = grossSales;
32         this.commissionRate = commissionRate;
33     }
34
35     // return first name
36     public String getFirstName() {return firstName
37
38     // return last name
39     public String getLastName() {return lastName;}
40
41     // return social security number
42     public String getSocialSecurityNumber() {retur
43
44     // set gross sales amount
45     public void setGrossSales(double grossSales) {
46         if (grossSales < 0.0) {
47             throw new IllegalArgumentException("Gros
48                 }
49
50         this.grossSales = grossSales;
51     }
52
53     // return gross sales amount
54     public double getGrossSales() {return grossSal
55
56     // set commission rate

```

```

57     public void setCommissionRate(double commissionRate) {
58         if (commissionRate <= 0.0 || commissionRate > 1.0)
59             throw new IllegalArgumentException(
60                 "Commission rate must be > 0.0 and <= 1.0");
61     }
62
63     this.commissionRate = commissionRate;
64 }
65
66 // return commission rate
67 public double getCommissionRate() {return commissionRate;}
68
69 // calculate earnings
70 public double earnings() {return commissionRate * grossSales;}
71
72 // return String representation of CommissionEmployee
73 @Override // indicates that this method overrides the method in the superclass
74 public String toString() {
75     return String.format("%s: %s %s\n%s: %s\n%s: %s\n",
76         "commission employee", firstName, lastName,
77         "social security number", socialSecurityNumber,
78         "gross sales", grossSales,
79         "commission rate", commissionRate);
80 }
81 }

```

Fig. 9.4

CommissionEmployee class represents an employee paid a percentage of gross sales.

Overview of Class

CommissionEmployee's Methods and Instance Variables

Class `CommissionEmployee`'s `public` services include a constructor (lines 12–33) and methods `earnings` (line 70) and `toString` (lines 73–80). Lines 36–42 declare `public` *get* methods for the class's `final` instance variables (declared in lines 5–7) `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 45–67 declare `public` *set* and *get* methods for the class's `grossSales` and `commissionRate` instance variables (declared in lines 8–9). The class declares its instance variables as `private`, so objects of other classes cannot directly access these variables.

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 15 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 18–20 and 23–26 validate the `grossSales` and `commissionRate` arguments. If these are valid (that is, the constructor does not throw an `IllegalArgumentException`), lines 28–32 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` (line 70) calculates a `CommissionEmployee`'s earnings. The method multiplies the `commissionRate` by the `grossSales` and returns the result.

Class CommissionEmployee's `toString` Method

Method `toString` (lines 73–80) 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 and the

object's so-called `hashCode` (see [Section 9.6](#)). 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.

@Override Annotation

Line 73 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` with a lowercase “s.” 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 can 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. 7.9](#) 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 6–7 instantiate a `CommissionEmployee` object and invoke `CommissionEmployee`'s constructor (lines 12–33 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 11–20 use `CommissionEmployee`'s *get* methods to retrieve the object's instance-variable values for output. Lines 22–23 invoke the object's `setGrossSales` and `setCommissionRate` methods to change the values of instance variables `grossSales` and `commissionRate`. Lines 25–26 output the `String` representation of the updated `CommissionEmployee`. When an object is output using the `%S` format specifier, the object's *to-String* 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 public class CommissionEmployeeTest {
4     public static void main(String[] args) {
5         // instantiate CommissionEmployee object
6         CommissionEmployee employee = new CommissionE
7             "Sue", "Jones", "222-22-2222", 10000, .06)
8
9         // get commission employee data
10        System.out.println("Employee information obta
11        System.out.printf("%n%s %s%n", "First name is
12            employee.getFirstName());
13        System.out.printf("%s %s%n", "Last name is",
14            employee.getLastName());
15        System.out.printf("%s %s%n", "Social security
16            employee.getSocialSecurityNumber());
17        System.out.printf("%s %.2f%n", "Gross sales i
18            employee.getGrossSales());
19        System.out.printf("%s %.2f%n", "Commission ra
```

```
20      employee.getCommissionRate());
21
22      employee.setGrossSales(5000);
23      employee.setCommissionRate(.1);
24
25      System.out.printf("%n%s:%n%n%s%n",
26      "Updated employee information obtained by
27      }
28  }
```

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 13–40) and methods `earnings` (lines 89–91) and `toString` (lines 94–102). Lines 43–86 declare public *get* and *set* methods for the class's private instance variables (declared in lines 5–10) `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
3 // a base salary in addition to commission.
4 public class BasePlusCommissionEmployee {
5     private final String firstName;
6     private final String lastName;
7     private final String socialSecurityNumber;
8     private double grossSales; // gross weekly sale
9     private double commissionRate; // commission pe
```

```

10     private double baseSalary; // base salary per w
11
12     // six-argument constructor
13     public BasePlusCommissionEmployee(String firstName
14         String socialSecurityNumber, double grossSal
15         double commissionRate, double baseSalary) {
16         // implicit call to Object's default constru
17
18         // if grossSales is invalid throw exception
19         if (grossSales < 0.0) {
20             throw new IllegalArgumentException("Gross
21                 }
22
23         // if commissionRate is invalid throw except
24         if (commissionRate <= 0.0 || commissionRate
25             throw new IllegalArgumentException(
26                 "Commission rate must be > 0.0 and <
27                 }
28
29         // if baseSalary is invalid throw exception
30         if (baseSalary < 0.0) {
31             throw new IllegalArgumentException("Base
32                 }
33
34         this.firstName = firstName;
35         this.lastName = lastName;
36         this.socialSecurityNumber = socialSecurityNu
37         this.grossSales = grossSales;
38         this.commissionRate = commissionRate;
39         this.baseSalary = baseSalary;
40     }
41
42     // return first name
43     public String getFirstName() {return firstName;
44
45     // return last name
46     public String getLastName() {return lastName;}
47
48     // return social security number
49     public String getSocialSecurityNumber() {return

```

```

50
51    // set gross sales amount
52    public void setGrossSales(double grossSales) {
53        if (grossSales < 0.0) {
54            throw new IllegalArgumentException("Gross
55                }
56
57        this.grossSales = grossSales;
58    }
59
60    // return gross sales amount
61    public double getGrossSales() {return grossSale
62
63        // set commission rate
64    public void setCommissionRate(double commission
65        if (commissionRate <= 0.0 || commissionRate
66            throw new IllegalArgumentException(
67                "Commission rate must be > 0.0 and <
68            }
69
70        this.commissionRate = commissionRate;
71    }
72
73    // return commission rate
74    public double getCommissionRate() {return commi
75
76    // set base salary
77    public void setBaseSalary(double baseSalary) {
78        if (baseSalary < 0.0) {
79            throw new IllegalArgumentException("Base
80        }
81
82        this.baseSalary = baseSalary;
83    }
84
85    // return base salary
86    public double getBaseSalary() {return baseSalar
87
88    // calculate earnings
89    public double earnings() {

```



```

90     public double getBaseSalary() {return baseSa
          91     }
          92
93     // return String representation of BasePlusComm
          94     @Override
          95     public String toString() {
          96         return String.format(
97             "%s: %s %s%n%s: %s%n%s: %.2f%n%s: %.2f%n%s
98             "base-salaried commission employee", first

          99             "social security number", socialSecurityNu
100             "gross sales", grossSales, "commission rat
101             "base salary", baseSalary);
          102     }
          103 }

```

Fig. 9.6

BasePlusCommissionEmployee class represents an employee who receives a base salary in addition to a commission.

Class BasePlusCommissionEmployee does *not* specify “extends Object” in line 4, so the class *implicitly* extends Object. Also, like class CommissionEmployee’s constructor (lines 12–33 of Fig. 9.4), class BasePlusCommissionEmployee’s constructor invokes class Object’s default constructor *implicitly*, as noted in the comment in line 16 of Fig. 9.6.

Class BasePlusCommissionEmployee’s earnings method (lines 89–91) 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 97).

Testing Class `BasePlusCommissionEmployee`

Figure 9.7 tests class `BasePlusCommissionEmployee`. Lines 7–9 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 14–25 use `BasePlusCommissionEmployee`'s *get* methods to retrieve the values of the object's instance variables for output. Line 27 invokes the object's `setBaseSalary` method to change the base salary. Method `setBaseSalary` (Fig. 9.6, lines 77–83) ensures that instance variable `baseSalary` is not assigned a negative value. Line 31 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     public static void main(String[] args) {
6         // instantiate BasePlusCommissionEmployee ob
7         BasePlusCommissionEmployee employee =
8             new BasePlusCommissionEmployee(
9                 "Bob", "Lewis", "333-33-3333", 5000, .04,
10
11         // get base-salaried commission employee dat
12             System.out.printf(
13                 "Employee information obtained by get met
14             System.out.printf("%s %s\n", "First name is"
15                 employee.getFirstName());
16             System.out.printf("%s %s\n", "Last name is",
17                 employee.getLastName());
18             System.out.printf("%s %s\n", "Social securit
19                 employee.getSocialSecurityNumber());
20             System.out.printf("%s %.2f\n", "Gross sales
21                 employee.getGrossSales());
22             System.out.printf("%s %.2f\n", "Commission r
23                 employee.getCommissionRate());
24             System.out.printf("%s %.2f\n", "Base salary
25                 employee.getBaseSalary());
26
27             employee.setBaseSalary(1000););
28
29             System.out.printf("%n%s:%n%n%s%n",
30             "Updated employee information obtained by
31                 employee.toString()
32             }
33     }

```

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.

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 `getFirstName` 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 4). Keyword `extends` (line 3) 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.

```
1 // Fig. 9.8: BasePlusCommissionEmployee.java
2 // private superclass members cannot be accessed i
3 public class BasePlusCommissionEmployee extends Co
4     private double baseSalary; // base salary per w
5
6     // six-argument constructor
7     public BasePlusCommissionEmployee(String firstN
8         String socialSecurityNumber, double grossSal
9         double commissionRate, double baseSalary) {
10         // explicit call to superclass CommissionEmp
11         super(firstName, lastName, socialSecurityNum
12             grossSales, commissionRate);
13
```

```

14      // if baseSalary is invalid throw exception
15      if (baseSalary < 0.0) {
16          throw new IllegalArgumentException("Base
17              }
18
19      this.baseSalary = baseSalary;
20      }
21
22      // set base salary
23  public void setBaseSalary(double baseSalary) {
24      if (baseSalary < 0.0) {
25          throw new IllegalArgumentException("Base
26              }
27
28      this.baseSalary = baseSalary;
29      }
30
31      // return base salary
32  public double getBaseSalary() {return baseSalary;
33
34      // calculate earnings
35      @Override
36      public double earnings() {
37          // not allowed: commissionRate and grossSales
38          return baseSalary + (commissionRate * grossSales);
39      }
40
41      // return String representation of BasePlusCommissionEmployee
42      @Override
43      public String toString() {
44          // not allowed: attempts to access private variables
45          return String.format(
46              "%s: %s %s\n%s: %s\n%s: %.2f%n%s: %.2f%n",
47              "base-salaried commission employee", firstName,
48              "social security number", socialSecurityNumber,
49              "gross sales", grossSales, "commission rate",
50              "base salary", baseSalary);
51      }
52  }

```



```
BasePlusCommissionEmployee.java:38: error: commission
    return baseSalary + (commissionRate * grossSale
                           ^
BasePlusCommissionEmployee.java:38: error: grossSales
    return baseSalary + (commissionRate * grossSale
                           ^

BasePlusCommissionEmployee.java:47: error: firstName
    "base-salaried commission employee", firstNa
                                         ^
BasePlusCommissionEmployee.java:47: error: lastName h
    "base-salaried commission employee", firstNa

BasePlusCommissionEmployee.java:48: error: socialSecu
    "social security number", socialSecurityNumb
                                         ^
BasePlusCommissionEmployee.java:49: error: grossSales
    "gross sales", grossSales, "commission rate"
                           ^
BasePlusCommissionEmployee.java:49: error: commission
    "gross sales", grossSales, "commission rate"
```

Fig. 9.8

private superclass members cannot be accessed in a subclass.

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 7–20), public methods inherited from `CommissionEmployee`, and methods `setBaseSalary` (lines 23–29), `getBaseSalary` (line 32), `earnings` (lines 35–39) and `toString` (lines 42–51). 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.

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 11–12 in `BasePlusCommissionEmployee`'s six-argument constructor (lines 7–20) explicitly call class `CommissionEmployee`'s five-argument constructor (declared at lines 12–33 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. The explicit superclass constructor call in lines 11–12 of [Fig. 9.8](#) must be the *first* statement in the constructor’s body.

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. 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 38 (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 47–49 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 38 could have called `getCommissionRate` and `getGrossSales` to access `CommissionEmployee`'s `private` instance variables `commissionRate` and `grossSales`, respectively. Lines 47–49 also could have used appropriate *get* methods to retrieve the values of the superclass's instance variables.


9.4.4

CommissionEmployee–Ba

sePlusCommissionEmplo yee 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 5–9 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 percent
```



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 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 38 of method `earnings` and lines 46–48 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 i
3 // variables from CommissionEmployee.
4
5 public class BasePlusCommissionEmployee extends Co
6     private double baseSalary; // base salary per w
7
8     // six-argument constructor
9     public BasePlusCommissionEmployee(String firstN
10         String socialSecurityNumber, double grossSal
11         double commissionRate, double baseSalary) {
12         super(firstName, lastName, socialSecurityNum
13             grossSales, commissionRate);
14
15         // if baseSalary is invalid throw exception
16         if (baseSalary < 0.0) {
17             throw new IllegalArgumentException("Base
18                 }
19
20         this.baseSalary = baseSalary;
21     }
22
23     // set base salary
24     public void setBaseSalary(double baseSalary) {
25         if (baseSalary < 0.0) {
26             throw new IllegalArgumentException("Base
27                 }
28
29         this.baseSalary = baseSalary;
30     }
31
32     // return base salary
33     public double getBaseSalary() {return baseSalar
34
35     // calculate earnings
36     @Override // indicates that this method overrid
37     public double earnings() {
```

```

38         return baseSalary + (commissionRate * grossS
39         }
40
41     // return String representation of BasePlusComm
42     @Override
43     public String toString() {
44         return String.format(
45             "%s: %s %s%n%s: %s%n%s: %.2f%n%s: %.2f%n%",
46             "base-salaried commission employee", firstN
47             "social security number", socialSecurityN
48             "gross sales", grossSales, "commission ra
49             "base salary", baseSalary);
50     }
51 }

```

Fig. 9.9

BasePlusCommissionEmployee inherits protected instance variables from CommissionEmployee.

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

When you create a BasePlusCommissionEmployee, 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 constructor, but *explicitly invokes*
it (lines 12–13) to initialize the
BasePlusCommissionEmployee instance variables
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

Class BasePlusCommissionEmployeeTest 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](#) (51 lines) is considerably shorter than that in [Fig. 9.6](#) (103
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

Avoid `protected` instance variables. 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 5–9) and provides public methods `getFirstName`, `getLastName`, `getSocialSecurityNumber`, `set-GrossSales`, `getGrossSales`, `setCommissionRate`, `getCommissionRate`, `earnings` and `toString` for manipulating these values. Methods `earnings` (lines 70–72) and `toString` (lines 75–82) 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 manipu
3 // private instance variables.
```

```

4 public class CommissionEmployee {
5     private final String firstName;
6     private final String lastName;
7     private final String socialSecurityNumber;
8     private double grossSales; // gross weekly sal
9     private double commissionRate; // commission p
10
11     // five-argument constructor
12     public CommissionEmployee(String firstName, St
13         String socialSecurityNumber, double grossSa
14         double commissionRate) {
15         // implicit call to Object constructor occu
16
17         // if grossSales is invalid throw exception
18         if (grossSales < 0.0) {
19             throw new IllegalArgumentException("Gros
20         }
21
22         // if commissionRate is invalid throw excep
23         if (commissionRate <= 0.0 || commissionRate
24             throw new IllegalArgumentException(
25                 "Commission rate must be > 0.0 and <
26         }
27
28         this.firstName = firstName;
29         this.lastName = lastName;
30         this.socialSecurityNumber = socialSecurityN
31         this.grossSales = grossSales;
32         this.commissionRate = commissionRate;
33     }
34
35     // return first name
36     public String getFirstName() {return firstName
37
38     // return last name
39     public String getLastName() {return lastName;}
40
41     // return social security number
42     public String getSocialSecurityNumber() {retur
43

```

```

44      // set gross sales amount
45  public void setGrossSales(double grossSales) {
46      if (grossSales < 0.0) {
47          throw new IllegalArgumentException("Gross sales amount must be > 0.0")
48      }
49
50      this.grossSales = grossSales;
51  }
52
53      // return gross sales amount
54  public double getGrossSales() {return grossSales;}
55
56      // set commission rate
57  public void setCommissionRate(double commissionRate) {
58      if (commissionRate <= 0.0 || commissionRate > 1.0) {
59          throw new IllegalArgumentException("Commission rate must be > 0.0 and <= 1.0")
60      }
61
62      this.commissionRate = commissionRate;
63  }
64
65      // return commission rate
66  public double getCommissionRate() {return commissionRate;}
67
68      // calculate earnings
69  public double earnings() {
70      return getCommissionRate() * getGrossSales();
71  }
72
73      // return String representation of CommissionEmployee
74  @Override
75  public String toString() {
76      return String.format("%s: %s %s\n%s: %s\n%s: %s\n%s",
77          "commission employee", getFirstName(), getLastName(),
78          "social security number", getSocialSecurityNumber(),
79          "gross sales", getGrossSales(),
80          "commission rate", getCommissionRate());
81  }
82
83  }

```



Fig. 9.10

`CommissionEmployee` class uses methods to manipulate its `private` instance variables.

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 36–37 of Fig. 9.11) and `toString` (lines 40–44) 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
3 // and accesses the superclass's private data via
```



```

4 // public methods.
5 public class BasePlusCommissionEmployee extends Co
6     private double baseSalary; // base salary per w
7
8     // six-argument constructor
9     public BasePlusCommissionEmployee(String firstN
10         String socialSecurityNumber, double grossSal
11         double commissionRate, double baseSalary) {
12         super(firstName, lastName, socialSecurityNum
13             grossSales, commissionRate);
14
15         // if baseSalary is invalid throw exception
16         if (baseSalary < 0.0) {
17             throw new IllegalArgumentException("Base
18                 }
19
20         this.baseSalary = baseSalary;
21     }
22
23     // set base salary
24     public void setBaseSalary(double baseSalary) {
25         if (baseSalary < 0.0) {
26             throw new IllegalArgumentException("Base
27                 }
28
29         this.baseSalary = baseSalary;
30     }
31
32     // return base salary
33     public double getBaseSalary() {return baseSalar
34
35     // calculate earnings
36     @Override
37     public double earnings() {return getBaseSalary(
38
39     // return String representation of BasePlusComm
40     @Override
41     public String toString() {
42         return String.format("%s %s%n%s: %.2f", "bas
43         super.toString(), "base salary", getBaseS

```

```
44     }  
45 }
```

Fig. 9.11

BasePlusCommissionEmployee class inherits from CommissionEmployee and accesses the superclass's private data via inherited public methods.

Class BasePlusCommissionEmployee's earnings Method

Method `earnings` (lines 36–37) overrides class `CommissionEmployee`'s `earnings` method (Fig. 9.10, lines 70–72) 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 37 of Fig. 9.11), 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 40–44) overrides `CommissionEmployee`'s `toString` method ([Fig. 9.10](#), lines 75–82) 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 43). `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.