Chapter 9
Object-Oriented Programming: Inheritance

Java How to Program, 11/e, Global Edition

Questions? E-mail paul.deitel@deitel.com

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.

OBJECTIVES

- 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 `BasePlus-CommissionEmployee` Class

OUTLINE

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

OUTLINE

9.5 Constructors in Subclasses

9.6 Class `Object`

9.7 Designing with Composition vs. Inheritance

9.8 Wrap-Up

9.1 Introduction

- Inheritance
  - **A new class** is created by **acquiring** an **existing class's members** and possibly embellishing them with **new or modified capabilities**.
  - Can save time during program development by basing new classes on existing proven and debugged high-quality software.
  - Increases the likelihood that a system will be implemented and maintained effectively.

9.1 Introduction (Cont.)

- 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.
  - Existing class is the superclass
  - New class is the subclass
- A subclass can be a superclass of future subclasses.
- A subclass can add its own fields and methods.
- 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 add behaviors that are specific to the subclass.
  - This is why inheritance is sometimes referred to as specialization.

9.1 Introduction (Cont.)

- 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.
- The Java class hierarchy begins with class `Object` (in package `java.lang`)
  - *Every* class in Java directly or indirectly extends (or "inherits from") `Object`.
- Java supports only single inheritance, in which each class is derived from exactly one direct superclass.

9.1 Introduction (Cont.)

- 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
- *Has-a* represents composition
  - In a *has-a* relationship, an object contains as members references to other objects

9.2 Superclasses and Subclasses

- Figure 9.1 lists several simple examples of superclasses and subclasses
  - Superclasses tend to be "more general" and subclasses "more specific."
- 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 typically **larger** than the set of objects represented by any of its subclasses.

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.

9.2 Superclasses and Subclasses (Cont.)

- A superclass exists in a hierarchical relationship with its subclasses.
- Fig. 9.2 shows a sample university community class hierarchy
  - Also called an inheritance hierarchy.

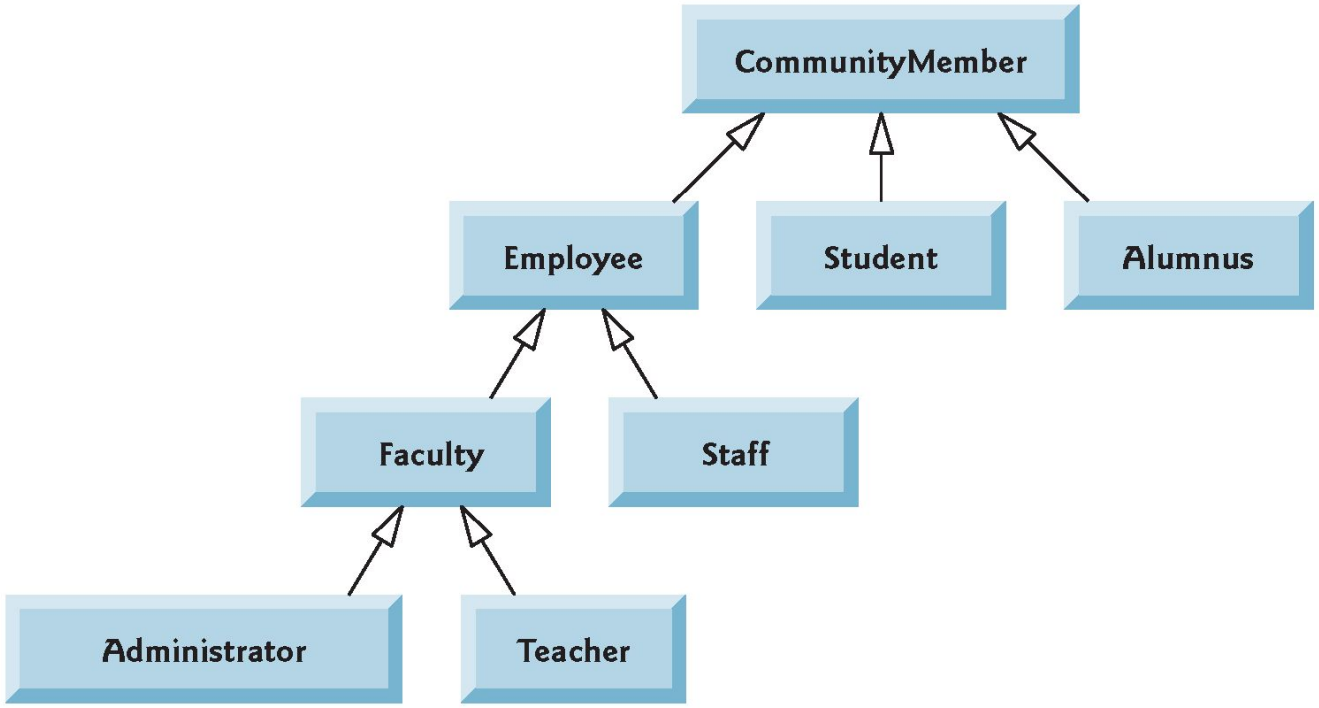

Fig. 9.2 | Inheritance hierarchy UML class diagram for university `CommunityMembers`.

9.2 Superclasses and Subclasses (Cont.)

- A superclass exists in a hierarchical relationship with its subclasses.
- Fig. 9.2 shows a sample university community class hierarchy
  - Also called an inheritance hierarchy.

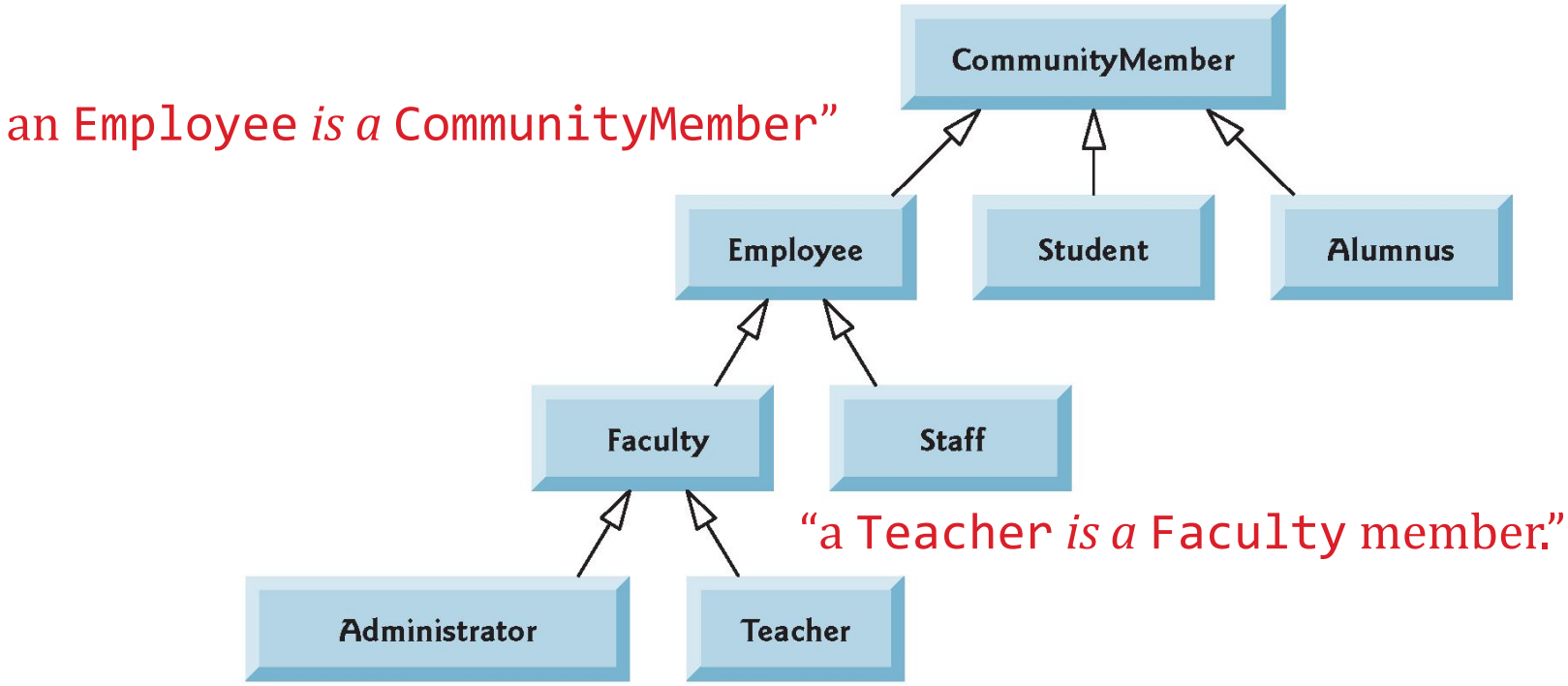

Fig. 9.2 | Inheritance hierarchy UML class diagram for university CommunityMembers.

9.2 Superclasses and Subclasses (Cont.)

- In the **inheritance hierarchy**:
  - Each arrow in the hierarchy represents an *is-a relationship.*
- Follow the arrows upward in the class hierarchy
  - an `Employee` *is a* `CommunityMember`"
  - "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.

9.2 Superclasses and Subclasses (Cont.)

- Fig. 9.3 shows a `Shape` inheritance hierarchy.
- Van follow the arrows from the bottom of the diagram to the topmost superclass in this class hierarchy to identify several *is-a* relationships.
  - A `Triangle` *is a* `TwoDimensionalShape` and *is a* `Shape`
  - A`Sphere` *is a* `ThreeDimensionalShape` and *is a* `Shape`.

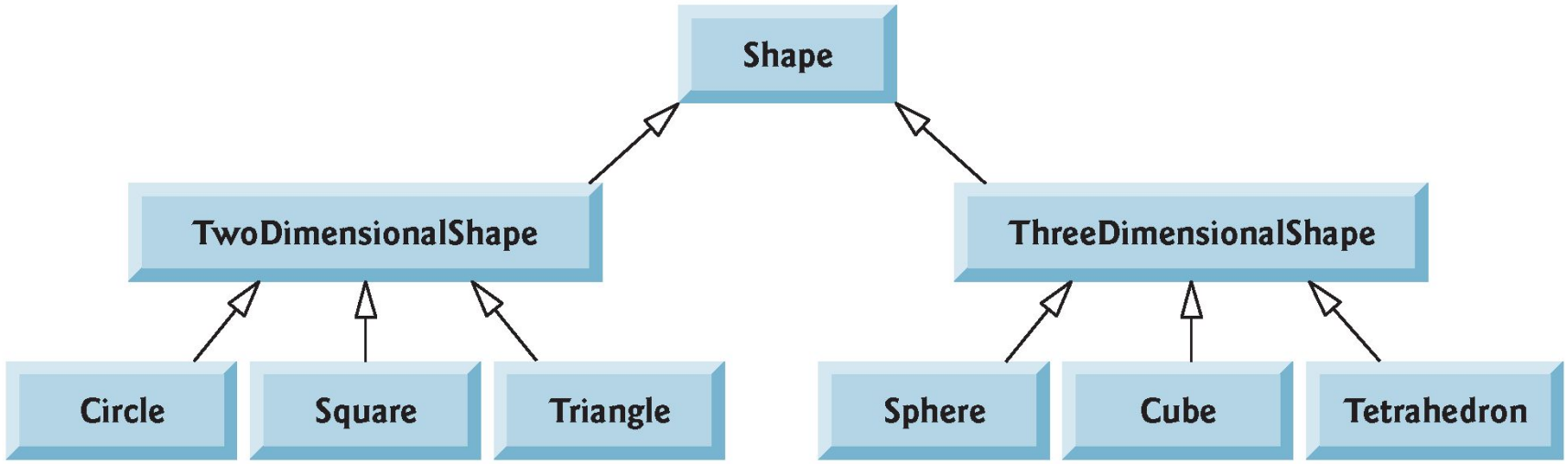

Fig. 9.3 | Inheritance hierarchy UML class diagram for `Shapes`.

9.2 Superclasses and Subclasses (Cont.)

- Not every class relationship is an inheritance relationship.
- *Has-a* relationship
  - Create classes by composition of existing classes.
  - 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`.

9.2 Superclasses and Subclasses (Cont.)

- Objects of all classes that extend a common superclass can be treated as objects of that superclass.
  - Commonality expressed in the members of the superclass.
- Inheritance issue
  - A subclass can inherit methods that it does not need or should not have.
  - Even when a superclass method is appropriate for a subclass, that subclass often needs a customized version of the method.
  - The subclass can override (redefine) the superclass method with an appropriate implementation.

9.3 `protected` Members

- 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.
- `protected` access is 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.

9.3 `protected` Members (Cont.)

- **A superclass's `private` members are *hidden* from its subclasses**
  - They 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.

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

- Inheritance hierarchy containing types of *employees* in a company's payroll application
- *Commission employees* are paid a percentage of their sales
- *Base-salaried commission employees* receive a base salary plus a percentage of their sales.

9.4.1 Creating and Using a `CommissionEmployee` Class

- Class `CommissionEmployee` (Fig. 9.4) `extends` class `Object` (from package `java.lang`).
  - `CommissionEmployee` inherits `Object`'s methods.
  - If you don't explicitly specify which class a new class extends, the class extends `Object` implicitly.

```
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     private final String firstName;
6     private final String lastName;
7     private final String socialSecurityNumber;
8     private double grossSales; // gross weekly sales
9     private double commissionRate; // commission percentage
10
```

Fig. 9.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 1 of 5.)

```
11      // five-argument constructor
12      public CommissionEmployee(String firstName, String lastName,
13         String socialSecurityNumber, double grossSales,
14         double commissionRate) {
15         // implicit call to Object's default constructor occurs here
16
17         // if grossSales is invalid throw exception
18         if (grossSales < 0.0) {
19            throw new IllegalArgumentException("Gross sales must be >= 0.0");
20         }
21
22         // if commissionRate is invalid throw exception
23         if (commissionRate <= 0.0 || commissionRate >= 1.0) {
24            throw new IllegalArgumentException(
25               "Commission rate must be > 0.0 and < 1.0");
26         }
27
28         this.firstName = firstName;
29         this.lastName = lastName;
30         this.socialSecurityNumber = socialSecurityNumber;
31         this.grossSales = grossSales;
32         this.commissionRate = commissionRate;
33      }
```

Fig. 9.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 2 of 5.)

```
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() {return socialSecurityNumber;}
43
44      // set gross sales amount
45      public void setGrossSales(double grossSales) {
46         if (grossSales < 0.0) {
47            throw new IllegalArgumentException("Gross sales must be >= 0.0");
48         }
49
50         this.grossSales = grossSales;
51      }
```

Fig. 9.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 3 of 5.)

```
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(
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
```

Fig. 9.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 4 of 5.)

```
69      // calculate earnings
70      public double earnings() {return commissionRate * grossSales;}
71
72      // return String representation of CommissionEmployee object
73      @Override // indicates that this method overrides a superclass method
74      public String toString() {
75         return String.format("%s: %s %s%n%s: %s%n%s: %.2f%n%s: %.2f",
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. (Part 5 of 5.)

9.4.1 Creating and Using a `CommissionEmployee` Class (Cont.)

- Constructors are *not* inherited.
- *The first task of a subclass constructor is to call its direct superclass's constructor* explicitly or implicitly
  - Ensures that the instance variables inherited from the superclass are initialized properly.
- If the code does not include an explicit call to the superclass constructor, Java implicitly calls the superclass's default or no-argument constructor.
- A class's default constructor calls the superclass's default or no-argument constructor.

9.4.1 Creating and Using a `CommissionEmployee` Class (Cont.)

- `toString` is one of the methods that *every* class inherits directly or indirectly from class `Object`.
  - Returns a `String` representing an object.
  - Called implicitly whenever an object must be converted to a `String` representation.
- Class `Object`'s `toString` method returns a `String` that includes the name of the object's class.
  - This is primarily a placeholder that can be *overridden* by a subclass to specify an appropriate `String` representation.

9.4.1 Creating and Using a `CommissionEmployee` Class (Cont.)

- To override a superclass method, a subclass must declare a method with the same signature as the superclass method
- `@Override` annotation
  - Indicates that a method should override a superclass method with the same signature.
  - If it does not, a compilation error occurs.

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.

```
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 CommissionEmployee(
7           "Sue", "Jones", "222-22-2222", 10000, .06);
8
9        // get commission employee data
10       System.out.println("Employee information obtained by get methods:");
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 number is",
16          employee.getSocialSecurityNumber());
17       System.out.printf("%s %.2f%n", "Gross sales is",
18          employee.getGrossSales());
19       System.out.printf("%s %.2f%n", "Commission rate is",
20          employee.getCommissionRate());
```

Fig. 9.5 | `CommissionEmployee` class test program. (Part 1 of 2.)

```
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 toString", employee);
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. (Part 2 of 2.)

9.4.2 Creating and Using a `BasePlus-CommissionEmployee` Class

- Class `BasePlusCommissionEmployee` (Fig. 9.6) contains a first name, last name, social security number, gross sales amount, commission rate *and* base salary.
  - All but the base salary are in common with class `CommissionEmployee`.
- Class `BasePlusCommissionEmployee`'s `public` services include a constructor, and methods `earnings`, `toString` and *get* and *set* for each instance variable
  - Most of these are in common with class `CommissionEmployee`.

```
1  // Fig. 9.6: BasePlusCommissionEmployee.java
2  // BasePlusCommissionEmployee class represents an employee who receives
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 sales
9     private double commissionRate; // commission percentage
10    private double baseSalary; // base salary per week
11
```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 1 of 7.)

```
12      // six-argument constructor
13      public BasePlusCommissionEmployee(String firstName, String lastName,
14         String socialSecurityNumber, double grossSales,
15         double commissionRate, double baseSalary) {
16         // implicit call to Object's default constructor occurs here
17
18         // if grossSales is invalid throw exception
19         if (grossSales < 0.0) {
20            throw new IllegalArgumentException("Gross sales must be >= 0.0");
21         }
22
23         // if commissionRate is invalid throw exception
24         if (commissionRate <= 0.0 || commissionRate >= 1.0) {
25            throw new IllegalArgumentException(
26               "Commission rate must be > 0.0 and < 1.0");
27         }
28
```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 2 of 7.)

```
29          // if baseSalary is invalid throw exception
30          if (baseSalary < 0.0) {
31             throw new IllegalArgumentException("Base salary must be >= 0.0");
32          }
33
34          this.firstName = firstName;
35          this.lastName = lastName;
36          this.socialSecurityNumber = socialSecurityNumber;
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
```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 3 of 7.)

```
48      // return social security number
49      public String getSocialSecurityNumber() {return socialSecurityNumber;}
50
51      // set gross sales amount
52      public void setGrossSales(double grossSales) {
53         if (grossSales < 0.0) {
54            throw new IllegalArgumentException("Gross sales must be >= 0.0");
55         }
56
57         this.grossSales = grossSales;
58      }
59
60      // return gross sales amount
61      public double getGrossSales() {return grossSales;}
62
```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 4 of 7.)

```
63      // set commission rate
64      public void setCommissionRate(double commissionRate) {
65         if (commissionRate <= 0.0 || commissionRate >= 1.0) {
66            throw new IllegalArgumentException(
67               "Commission rate must be > 0.0 and < 1.0");
68         }
69
70         this.commissionRate = commissionRate;
71      }
72
73      // return commission rate
74      public double getCommissionRate() {return commissionRate;}
75
```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 5 of 7.)

```
76      // set base salary
77      public void setBaseSalary(double baseSalary) {
78         if (baseSalary < 0.0) {
79            throw new IllegalArgumentException("Base salary must be >= 0.0");
80         }
81
82         this.baseSalary = baseSalary;
83      }
84
85      // return base salary
86      public double getBaseSalary() {return baseSalary;}
87
```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 6 of 7.)

```
88      // calculate earnings
89      public double earnings() {
90         return baseSalary + (commissionRate * grossSales);
91      }
92
93      // return String representation of BasePlusCommissionEmployee
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: %.2f",
98            "base-salaried commission employee", firstName, lastName,
99            "social security number", socialSecurityNumber,
100           "gross sales", grossSales, "commission rate", commissionRate,
101           "base salary", baseSalary);
102     }
103  }
```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 7 of 7.)

9.4.2 Creating and Using a `BasePlus-CommissionEmployee` Class (Cont.)

- Class `BasePlusCommissionEmployee` does *not* specify "`extends Object`"
  - Implicitly extends `Object`.
- `BasePlusCommissionEmployee`'s constructor invokes class `Object`'s default constructor *implicitly*.

```
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 object
7        BasePlusCommissionEmployee employee =
8           new BasePlusCommissionEmployee(
9           "Bob", "Lewis", "333-33-3333", 5000, .04, 300);
10
11       // get base-salaried commission employee data
12       System.out.printf(
13          "Employee information obtained by get methods:%n");
14       System.out.printf("%s %s%n", "First name is",
15          employee.getFirstName());
```

Fig. 9.7 | BasePlusCommissionEmployee test program. (Part 1 of 3.)

```
16          System.out.printf("%s %s%n", "Last name is",
17             employee.getLastName());
18          System.out.printf("%s %s%n", "Social security number is",
19             employee.getSocialSecurityNumber());
20          System.out.printf("%s %.2f%n", "Gross sales is",
21             employee.getGrossSales());
22          System.out.printf("%s %.2f%n", "Commission rate is",
23             employee.getCommissionRate());
24          System.out.printf("%s %.2f%n", "Base salary is",
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 toString",
31              employee.toString());
32       }
33    }
```

Fig. 9.7 | `BasePlusCommissionEmployee` test program. (Part 2 of 3.)

```
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 3 of 3.)

9.4.2 Creating and Using a `BasePlus-CommissionEmployee` Class (Cont.)

- Much of `BasePlusCommissionEmployee`'s code is *similar*, or *identical*, to that of `CommissionEmployee`.
- `private` instance variables `firstName` and `lastName` and methods `setFirstName`, `getFirstName`, `setLastName` and `getLastName` are identical.
  - Both classes also contain corresponding *get* and *set* methods.
- The constructors are almost identical
  - `BasePlusCommissionEmployee`'s constructor also sets the `baseSalary`.
- The `toString` methods are *almost* identical
  - `BasePlusCommissionEmployee`'s `toString` also outputs instance variable `baseSalary`

9.4.2 Creating and Using a `BasePlus-CommissionEmployee` Class (Cont.)

- We literally *copied* `CommissionEmployee`'s code, *pasted* it into `BasePlusCommissionEmployee`, then modified the new class to include a base salary and methods that manipulate the base salary.
  - This "*copy-and-paste*" *approach* is often error prone and time consuming.
  - It spreads copies of the same code throughout a system, creating a code-maintenance problems—changes to the code would need to be made in multiple classes.

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

- Class `BasePlusCommissionEmployee` class *extends* class `CommissionEmployee`
- A `BasePlusCommissionEmployee` object *is a* `CommissionEmployee`
  - Inheritance passes on class `CommissionEmployee`'s capabilities.
- Class `BasePlusCommissionEmployee` also has instance variable `baseSalary`.
- Subclass `BasePlusCommissionEmployee` inherits `CommissionEmployee`'s instance variables and methods
  - Only `CommissionEmployee`'s `public` and `protected` members are directly accessible in the subclass.

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 in a subclass.
3  public class BasePlusCommissionEmployee extends CommissionEmployee {
4     private double baseSalary; // base salary per week
5
6     // six-argument constructor
7     public BasePlusCommissionEmployee(String firstName, String lastName,
8        String socialSecurityNumber, double grossSales,
9        double commissionRate, double baseSalary) {
10       // explicit call to superclass CommissionEmployee constructor
11       super(firstName, lastName, socialSecurityNumber,
12          grossSales, commissionRate);
13
14       // if baseSalary is invalid throw exception
15       if (baseSalary < 0.0) {
16          throw new IllegalArgumentException("Base salary must be >= 0.0");
17       }
18
19       this.baseSalary = baseSalary;
20    }
```

Fig. 9.8 | `private` superclass members cannot be accessed in a subclass. (Part 1 of 5.)

```
21
22      // set base salary
23      public void setBaseSalary(double baseSalary) {
24         if (baseSalary < 0.0) {
25            throw new IllegalArgumentException("Base salary must be >= 0.0");
26         }
27
28         this.baseSalary = baseSalary;
29      }
30
31      // return base salary
32      public double getBaseSalary() {return baseSalary;}
33
```

Fig. 9.8 | `private` superclass members cannot be accessed in a subclass. (Part 2 of 5.)

```
34      // calculate earnings
35      @Override
36      public double earnings() {
37         // not allowed: commissionRate and grossSales private in superclass
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 superclass members
45         return String.format(
46            "%s: %s %s%n%s: %s%n%s: %.2f%n%s: %.2f%n%s: %.2f",
47            "base-salaried commission employee", firstName, lastName,
48            "social security number", socialSecurityNumber,
49            "gross sales", grossSales, "commission rate", commissionRate,
50            "base salary", baseSalary);
51      }
52   }
```

Fig. 9.8 | `private` superclass members cannot be accessed in a subclass. (Part 3 of 5.)

```
BasePlusCommissionEmployee.java:38: error: commissionRate has private access
in CommissionEmployee
      return baseSalary + (commissionRate * grossSales);
                           ^
BasePlusCommissionEmployee.java:38: error: grossSales has private access in
CommissionEmployee
      return baseSalary + (commissionRate * grossSales);
                                            ^

BasePlusCommissionEmployee.java:47: error: firstName has private access in
CommissionEmployee
         "base-salaried commission employee", firstName, lastName,
                                              ^
BasePlusCommissionEmployee.java:47: error: lastName has private access in
CommissionEmployee
         "base-salaried commission employee", firstName, lastName,
                                                         ^
```

Fig. 9.8 | `private` superclass members cannot be accessed in a subclass. (Part 4 of 5.)

```
BasePlusCommissionEmployee.java:48: error: socialSecurityNumber has private
access in CommissionEmployee
         "social security number", socialSecurityNumber,
                                   ^
BasePlusCommissionEmployee.java:49: error: grossSales has private access in
CommissionEmployee
         "gross sales", grossSales, "commission rate", commissionRate,
                        ^
BasePlusCommissionEmployee.java:49: 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 5 of 5.)

9.4.3 Creating a `CommissionEmployee–BasePlusCommissionEmployee` Inheritance Hierarchy (Cont.)

- Each subclass constructor must implicitly or explicitly call one of its superclass's constructors to initialize the instance variables inherited from the superclass.
  - Superclass constructor call syntax—keyword `super`, followed by a set of parentheses containing the superclass constructor arguments.
  - Must be the *first* statement in the constructor's body.
- If the subclass 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.
- You can explicitly use `super()` to call the superclass's no-argument or default constructor, 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.

9.4.3 Creating a `CommissionEmployee–BasePlusCommissionEmployee` Inheritance Hierarchy (Cont.)

- Compilation errors occur when the subclass attempts to access the superclass's `private` instance variables.
- These lines 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 a subclass to directly access superclass instance variables, we can declare those members as `protected` in the superclass.
- New `CommissionEmployee` class modified only the instance variable declarations of Fig. 9.4 as follows:

```
protected final String firstName;
protected final String lastName;
protected final String socialSecurityNumber;
protected double grossSales;
protected double commissionRate;
```

- 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.

9.4.4 `CommissionEmployee–BasePlus-CommissionEmployee` Inheritance Hierarchy Using `protected` Instance Variables (Cont.)

- Class `BasePlusCommissionEmployee` (Fig. 9.9) extends the new version of class `CommissionEmployee` with `protected` instance variables.
  - These variables are now `protected` members of `BasePlusCommissionEmployee`.
- If another class extends this version of class `BasePlusCommissionEmployee`, the new subclass also can access the `protected` members.
- The source code in Fig. 9.9 is considerably shorter than that in Fig. 9.6
  - Most of the functionality is now inherited from `CommissionEmployee`
  - There is now only one copy of the functionality.
  - Code is easier to maintain, modify and debug—the code related to a `CommissionEmployee` exists only in that class.

```
1  // Fig. 9.9: BasePlusCommissionEmployee.java
2  // BasePlusCommissionEmployee inherits protected instance
3  // variables from CommissionEmployee.
4
5  public class BasePlusCommissionEmployee extends CommissionEmployee {
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       super(firstName, lastName, socialSecurityNumber,
13          grossSales, commissionRate);
14
15       // if baseSalary is invalid throw exception
16       if (baseSalary < 0.0) {
17          throw new IllegalArgumentException("Base salary must be >= 0.0");
18       }
19
20       this.baseSalary = baseSalary;
21    }
```

Fig. 9.9 | BasePlusCommissionEmployee inherits protected instance variables from CommissionEmployee. (Part 1 of 3.)

```
22
23      // set base salary
24      public void setBaseSalary(double baseSalary) {
25         if (baseSalary < 0.0) {
26            throw new IllegalArgumentException("Base salary must be >= 0.0");
27         }
28
29         this.baseSalary = baseSalary;
30      }
31
32      // return base salary
33      public double getBaseSalary() {return baseSalary;}
34
```

Fig. 9.9 | `BasePlusCommissionEmployee` inherits `protected` instance variables from `CommissionEmployee`. (Part 2 of 3.)

```
35      // calculate earnings
36      @Override // indicates that this method overrides a superclass method
37      public double earnings() {
38         return baseSalary + (commissionRate * grossSales);
39      }
40
41      // return String representation of BasePlusCommissionEmployee
42      @Override
43      public String toString() {
44         return String.format(
45            "%s: %s %s%n%s: %s%n%s: %.2f%n%s: %.2f%n%s: %.2f",
46            "base-salaried commission employee", firstName, lastName,
47            "social security number", socialSecurityNumber,
48            "gross sales", grossSales, "commission rate", commissionRate,
49            "base salary", baseSalary);
50      }
51   }
```

Fig. 9.9 | `BasePlusCommissionEmployee` inherits `protected` instance variables from `CommissionEmployee`. (Part 3 of 3.)

9.4.4 `CommissionEmployee–BasePlus-CommissionEmployee` Inheritance Hierarchy Using `protected` Instance Variables (Cont.)

- Inheriting `protected` instance variables enables direct access to the variables by subclasses.
- In most cases, it's better to use `private` instance variables to encourage proper software engineering.
  - Code will be easier to maintain, modify and debug.

9.4.4 `CommissionEmployee–BasePlus-CommissionEmployee` Inheritance Hierarchy Using `protected` Instance Variables (Cont.)

- Using `protected` instance variables creates several potential problems.
- The subclass object can set an inherited variable's value directly without using a *set method.*
  - A subclass object can assign an invalid value to the variable
- Subclass methods are more likely to be written so that they depend on the superclass's data implementation.
  - Subclasses should depend only on the superclass services and not on the superclass data implementation.

9.4.4 CommissionEmployee–BasePlus-CommissionEmployee Inheritance Hierarchy Using protected Instance Variables (Cont.)

- With `protected` instance variables in the superclass, we may need to modify all the subclasses of the superclass if the superclass implementation changes.
  - 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.
  - If the superclass services change, we must reimplement our subclasses.
- 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–BasePlus-CommissionEmployee` Inheritance Hierarchy Using `private` Instance Variables

- Class `CommissionEmployee` declares instance variables `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate` as *`private`* and provides `public` methods for manipulating these values.

9.4.5 `CommissionEmployee–BasePlus-CommissionEmployee` Inheritance Hierarchy Using `private` Instance Variables (Cont.)

- `CommissionEmployee` methods `earnings` and `toString` use the class's *get* methods to obtain the values of its instance variables.
  - If we decide to change the internal representation of the data (e.g., variable names) 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.
- Subclass `BasePlusCommissionEmployee` inherits `Commission-Employee`'s non-`private` methods and can access the `private` superclass members via those methods.

```
1  // Fig. 9.10: CommissionEmployee.java
2  // CommissionEmployee class uses methods to manipulate its
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 sales
9     private double commissionRate; // commission percentage
10
11    // five-argument constructor
12    public CommissionEmployee(String firstName, String lastName,
13       String socialSecurityNumber, double grossSales,
14       double commissionRate) {
15       // implicit call to Object constructor occurs here
16
17       // if grossSales is invalid throw exception
18       if (grossSales < 0.0) {
19          throw new IllegalArgumentException("Gross sales must be >= 0.0");
20       }
```

Fig. 9.10 | `CommissionEmployee` class uses methods to manipulate its `private` instance variables. (Part 1 of 5.)

```
21
22          // if commissionRate is invalid throw exception
23          if (commissionRate <= 0.0 || commissionRate >= 1.0) {
24             throw new IllegalArgumentException(
25                "Commission rate must be > 0.0 and < 1.0");
26          }
27
28          this.firstName = firstName;
29          this.lastName = lastName;
30          this.socialSecurityNumber = socialSecurityNumber;
31          this.grossSales = grossSales;
32          this.commissionRate = commissionRate;
33       }
34
```

Fig. 9.10 | `CommissionEmployee` class uses methods to manipulate its `private` instance variables. (Part 2 of 5.)

```
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() {return socialSecurityNumber;}
43
44      // set gross sales amount
45      public void setGrossSales(double grossSales) {
46         if (grossSales < 0.0) {
47            throw new IllegalArgumentException("Gross sales must be >= 0.0");
48         }
49
50         this.grossSales = grossSales;
51      }
52
53      // return gross sales amount
54      public double getGrossSales() {return grossSales;}
```

Fig. 9.10 | `CommissionEmployee` class uses methods to manipulate its `private` instance variables. (Part 3 of 5.)

```
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
```

Fig. 9.10 | `CommissionEmployee` class uses methods to manipulate its `private` instance variables. (Part 4 of 5.)

```
69      // calculate earnings
70      public double earnings() {
71         return getCommissionRate() * getGrossSales();
72      }
73
74      // return String representation of CommissionEmployee object
75      @Override
76      public String toString() {
77         return String.format("%s: %s %s%n%s: %s%n%s: %.2f%n%s: %.2f",
78            "commission employee", getFirstName(), getLastName(),
79            "social security number", getSocialSecurityNumber(),
80            "gross sales", getGrossSales(),
81            "commission rate", getCommissionRate());
82      }
83   }
```

Fig. 9.10 | `CommissionEmployee` class uses methods to manipulate its `private` instance variables. (Part 5 of 5.)

9.4.5 `CommissionEmployee–BasePlus-CommissionEmployee` Inheritance Hierarchy Using `private` Instance Variables (Cont.)

- Class `BasePlusCommissionEmployee` (Fig. 9.11) has several changes that distinguish it from Fig. 9.9.
- Methods `earnings` and `toString` each invoke their superclass versions and do not access instance variables directly.

```
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  public class BasePlusCommissionEmployee extends CommissionEmployee {
6     private double baseSalary; // base salary per week
7
```

Fig. 9.11 | `BasePlusCommissionEmployee` class inherits from `CommissionEmployee` and accesses the superclass's `private` data via inherited `public` methods. (Part 1 of 4.)

```
 8      // six-argument constructor
 9      public BasePlusCommissionEmployee(String firstName, String lastName,
10         String socialSecurityNumber, double grossSales,
11         double commissionRate, double baseSalary) {
12         super(firstName, lastName, socialSecurityNumber,
13            grossSales, commissionRate);
14
15         // if baseSalary is invalid throw exception
16         if (baseSalary < 0.0) {
17            throw new IllegalArgumentException("Base salary must be >= 0.0");
18         }
19
20         this.baseSalary = baseSalary;
21      }
22
```

Fig. 9.11 | `BasePlusCommissionEmployee` class inherits from `CommissionEmployee` and accesses the superclass's `private` data via inherited `public` methods. (Part 2 of 4.)

```
23      // set base salary
24      public void setBaseSalary(double baseSalary) {
25         if (baseSalary < 0.0) {
26            throw new IllegalArgumentException("Base salary must be >= 0.0");
27         }
28
29         this.baseSalary = baseSalary;
30      }
31
32      // return base salary
33      public double getBaseSalary() {return baseSalary;}
34
```

Fig. 9.11 | `BasePlusCommissionEmployee` class inherits from `CommissionEmployee` and accesses the superclass's `private` data via inherited `public` methods. (Part 3 of 4.)

```
35      // calculate earnings
36      @Override
37      public double earnings() {return getBaseSalary() + super.earnings();}
38
39      // return String representation of BasePlusCommissionEmployee
40      @Override
41      public String toString() {
42         return String.format("%s %s%n%s: %.2f", "base-salaried",
43            super.toString(), "base salary", getBaseSalary());
44      }
45   }
```

Fig. 9.11 | `BasePlusCommissionEmployee` class inherits from `CommissionEmployee` and accesses the superclass's `private` data via inherited `public` methods. (Part 4 of 4.)

9.4.5 `CommissionEmployee–BasePlus-CommissionEmployee` Inheritance Hierarchy Using `private` Instance Variables (Cont.)

- Method `earnings` overrides class the superclass's `earnings` method.
- The new version calls `CommissionEmployee`'s `earnings` method with `super.earnings()`.
  - Obtains the earnings based on commission alone
- Placing the keyword `super` and a dot (`.`) separator before the superclass method name invokes the superclass version of an overridden method.
- 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.

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.

9.4.5 CommissionEmployee–BasePlus-CommissionEmployee Inheritance Hierarchy Using private Instance Variables (Cont.)

- BasePlusCommissionEmployee's toString method overrides class CommissionEmployee's toString method.
- The new version creates part of the String representation by calling CommissionEmployee's toString method with the expression super.toString().

9.5 Constructors in Subclasses

- Instantiating a subclass object begins a chain of constructor calls
  - 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
- If the superclass is derived from another class, 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.
- Original subclass constructor's body finishes executing *last*.
- Each superclass's constructor manipulates the superclass instance variables that the subclass object inherits.

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`

- All classes in Java inherit directly or indirectly from class `Object`, so its 11 methods are inherited by all other classes.
- Figure 9.12 summarizes `Object`'s methods.
- Every array has an overridden `clone` method that copies the array.
  - If the array stores references to objects, the objects are not copied—a *shallow copy* is performed.

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 `http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html`. 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`.

Fig. 9.12 | `Object` methods. (Part 1 of 4.)

Method	Description
`hashCode`	Hashcodes are `int` values used for high-speed storage and retrieval of information stored in a hashtable data structure (see Section 16.10). This method is also called as part of `Object`'s default `toString` method implementation.
`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.

Fig. 9.12 | `Object` methods. (Part 2 of 4.)

Method	Description
`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`.

Fig. 9.12 | `Object` methods. (Part 3 of 4.)

Method	Description
`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 4 of 4.)

9.7 Designing with Composition vs. Inheritance

- There's much discussion in the software engineering community about the relative merits of composition and inheritance
- Each has its own place, but inheritance is often overused and composition is more appropriate in many cases
- A mix of composition and inheritance often is a reasonable design approach, as you'll see in Exercise 9.16

Software Engineering Observation 9.10

As a college student, you learn tools for creating solutions. In industry, problems are larger and more complex than you see in college courses. Often the demands of the problem you're solving will be unique and may require you to rethink the proper way to use the tools at your disposal. As a college student, you tend to work on problems yourself. In industry, problem solving often requires interaction among many colleagues. Rarely will you be able to get everyone on a project to agree on the "right" approach to a solution. Also, rarely will any particular approach be "perfect." You'll often compare the relative merits of different approaches, as we do in this section.

9.7 Designing with Composition vs. Inheritance (cont.)

Inheritance-Based Designs

- Inheritance creates tight coupling among the classes in a hierarchy
  - Each subclass typically depends on its direct or indirect superclasses' implementations
- Changes in superclass implementation can affect the behavior of subclasses, often in subtle ways
- Tightly coupled designs are more difficult to modify than those in loosely coupled, composition-based designs
- Change is the rule rather than the exception—this encourages composition

9.7 Designing with Composition vs. Inheritance (cont.)

- In general, use inheritance only for true *is-a* relationships in which you can assign a subclass object to a superclass reference
- When you invoke a method via a superclass reference to a subclass object, the subclass's corresponding method executes
  - This is called polymorphic behavior, which we explore in Chapter 10

Software Engineering Observation 9.11

Some of the difficulties with inheritance occur on large projects where different classes in the hierarchy are controlled by different people. An inheritance hierarchy is less problematic when it's entirely under one person's control.

9.7 Designing with Composition vs. Inheritance (cont.)

Composition-Based Designs

- Composition is loosely coupled
- When you compose a reference as an instance variable of a class, it's part of the class's implementation details that are hidden from the class's client code
- If the reference's class type changes, you may need to make changes to the composing class's internal details, but those changes do not affect the client code

9.7 Designing with Composition vs. Inheritance (cont.)

Composition-Based Designs

- In addition, inheritance is done at compile time
- Composition is more flexible—it, too, can be done at compile time, but it also can be done at execution time because non-final references to composed objects can be modified
  - We call this dynamic composition
- This is another aspect of loose coupling—if the reference is of a superclass type, you can replace the referenced object with an object of *any* type that has an *is-a* relationship with the reference's class type

9.7 Designing with Composition vs. Inheritance (cont.)

Composition-Based Designs

- When you use a composition approach instead of inheritance, you'll typically create a larger number of smaller classes, each focused on one responsibility
- Smaller classes generally are easier to test, debug and modify
- Java does not offer multiple inheritance—each class in Java may extend only one class
- However, a new class may reuse the capabilities of one or more other classes by composition. As you'll see in Chapter 10, we can get many of multiple inheritance's benefits by implementing multiple interfaces

Performance Tip 9.1

A potential disadvantage of composition is that it typically requires more objects at runtime, which might negatively impact garbage-collection and virtual-memory performance. Virtual-memory architectures and performance issues are typically discussed in operating systems courses.

Software Engineering Observation 9.12

A `public` method of a composing class can call a method of a composed object to perform a task for the benefit of the composing class's clients. This is known as forwarding the method call and is a common way to reuse a class's capabilities via composition rather than inheritance.

Software Engineering Observation 9.13

When implementing a new class and choosing whether you should reuse an existing class via inheritance or composition, use composition if the existing class's `public` methods should not be part of the new class's `public` interface.

9.7 Designing with Composition vs. Inheritance (cont.)

- ***Recommended Exercises***
- Exercise 9.3 asks you to reimplement this chapter's `CommissionEmployee–BasePlusCommissionEmployee` hierarchy using composition, rather than inheritance.
- Exercise 9.16 asks you to reimplement the hierarchy using a combination of composition and inheritance in which you'll see the benefits of composition's loose coupling.