



Chapter 9

Object-Oriented Programming: Inheritance



Objectives

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



9.1 Introduction

▶ Inheritance

- A form of software **reuse** in which a new class is created by absorbing an existing class's members and 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.
- Reduces 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**
- ▶ Each 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.

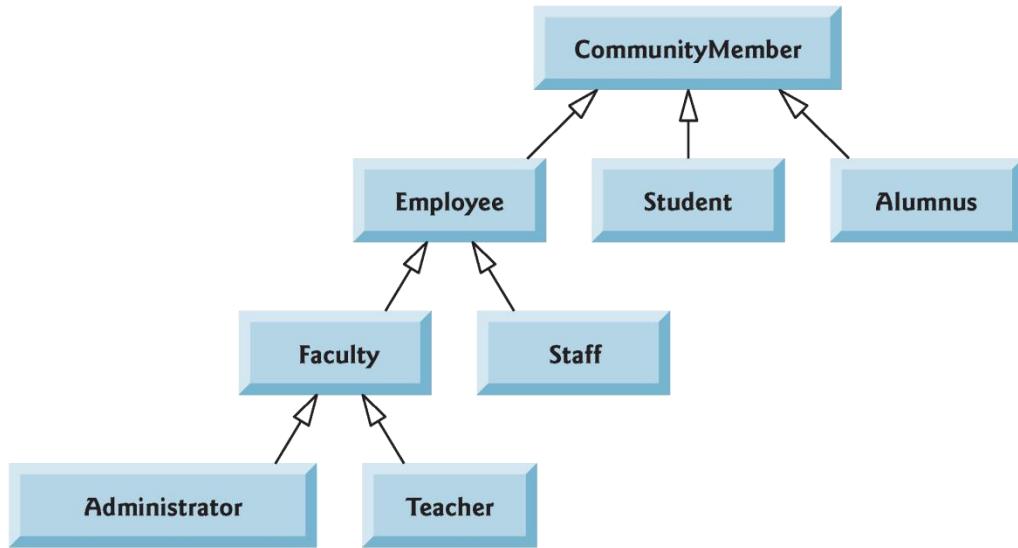


Fig. 9.2 | Inheritance hierarchy for university **CommunityMembers**.

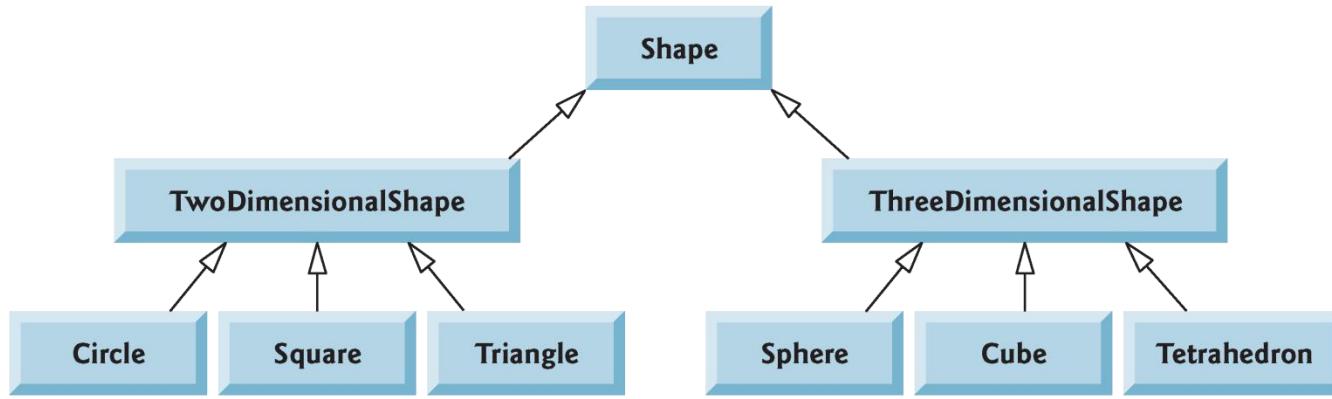


Fig. 9.3 | Inheritance hierarchy for Shapes.



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

- ▶ **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** in its subclasses
- ▶ 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 method can be accessed from the subclass by preceding the superclass method name with keyword **super** and a dot (.) separator.



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 {
6     private String firstName;
7     private String lastName;
8     private String socialSecurityNumber;
9     private double grossSales; // gross weekly sales
10    private double commissionRate; // commission percentage
11
12    // five-argument constructor
13    public CommissionEmployee( String first, String last, String ssn,
14        double sales, double rate )
15    {
16        // implicit call to Object constructor occurs here
17        firstName = first;
18        lastName = last;
19        socialSecurityNumber = ssn;
20        setGrossSales( sales ); // validate and store gross sales
21        setCommissionRate( rate ); // validate and store commission rate
22    } // end five-argument CommissionEmployee constructor
```

Fig. 9.4 | CommissionEmployee class represents an employee paid a percentage of gross sales. (Part I of 6.)



```
23
24     // set first name
25     public void setFirstName( String first )
26     {
27         firstName = first; // should validate
28     } // end method setFirstName
29
30     // return first name
31     public String getFirstName()
32     {
33         return firstName;
34     } // end method getFirstName
35
36     // set last name
37     public void setLastName( String last )
38     {
39         lastName = last; // should validate
40     } // end method setLastName
41
```

Fig. 9.4 | CommissionEmployee class represents an employee paid a percentage of gross sales. (Part 2 of 6.)



```
42     // return last name
43     public String getLastname()
44     {
45         return lastName;
46     } // end method getLastname
47
48     // set social security number
49     public void setSocialSecurityNumber( String ssn )
50     {
51         socialSecurityNumber = ssn; // should validate
52     } // end method setSocialSecurityNumber
53
54     // return social security number
55     public String getSocialSecurityNumber()
56     {
57         return socialSecurityNumber;
58     } // end method getSocialSecurityNumber
59
```

Fig. 9.4 | CommissionEmployee class represents an employee paid a percentage of gross sales. (Part 3 of 6.)



```
60 // set gross sales amount
61 public void setGrossSales( double sales )
62 {
63     if ( sales >= 0.0 )
64         grossSales = sales;
65     else
66         throw new IllegalArgumentException(
67             "Gross sales must be >= 0.0" );
68 } // end method setGrossSales
69
70 // return gross sales amount
71 public double getGrossSales()
72 {
73     return grossSales;
74 } // end method getGrossSales
75
```

Fig. 9.4 | CommissionEmployee class represents an employee paid a percentage of gross sales. (Part 4 of 6.)



```
76 // set commission rate
77 public void setCommissionRate( double rate )
78 {
79     if ( rate > 0.0 && rate < 1.0 )
80         commissionRate = rate;
81     else
82         throw new IllegalArgumentException(
83             "Commission rate must be > 0.0 and < 1.0" );
84 } // end method setCommissionRate
85
86 // return commission rate
87 public double getCommissionRate()
88 {
89     return commissionRate;
90 } // end method getCommissionRate
91
92 // calculate earnings
93 public double earnings()
94 {
95     return commissionRate * grossSales;
96 } // end method earnings
97
```

Fig. 9.4 | CommissionEmployee class represents an employee paid a percentage of gross sales. (Part 5 of 6.)



```
98 // return String representation of CommissionEmployee object
99 @Override // indicates that this method overrides a superclass method
100 public String toString()
101 {
102     return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
103         "commission employee", firstName, lastName,
104         "social security number", socialSecurityNumber,
105         "gross sales", grossSales,
106         "commission rate", commissionRate );
107 } // end method toString
108 } // end class CommissionEmployee
```

Fig. 9.4 | CommissionEmployee class represents an employee paid a percentage of gross sales. (Part 6 of 6.)



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.



Common Programming Error 9.1

Using an incorrect method signature when attempting to override a superclass method causes an unintentional method overload that can lead to subtle logic errors.



Error-Prevention Tip 9.1

Declare overridden methods with the @Override annotation 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.



Common Programming Error 9.2

It's a syntax error to override a method with a more restricted access modifier—a `public` method of the superclass cannot become a `protected` or `private` method in the subclass; a `protected` method of the superclass cannot become a `private` method in the subclass. Doing so would break the is-a relationship in which it's required that all subclass objects be able to respond to method calls that are made to `public` methods declared in the superclass. If a `public` method, for example, 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
4 public class CommissionEmployeeTest
{
5     public static void main( String[] args )
6     {
7         // instantiate CommissionEmployee object
8         CommissionEmployee employee = new CommissionEmployee(
9             "Sue", "Jones", "222-22-2222", 10000, .06 );
10
11
12     // get commission employee data
13     System.out.println(
14         "Employee information obtained by get methods: \n" );
15     System.out.printf( "%s %s\n", "First name is",
16         employee.getFirstName() );
17     System.out.printf( "%s %s\n", "Last name is",
18         employee.getLastName() );
19     System.out.printf( "%s %s\n", "Social security number is",
20         employee.getSocialSecurityNumber() );
21     System.out.printf( "%s %.2f\n", "Gross sales is",
22         employee.getGrossSales() );
23     System.out.printf( "%s %.2f\n", "Commission rate is",
24         employee.getCommissionRate() );
```

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



```
25  
26     employee.setGrossSales( 500 ); // set gross sales  
27     employee.setCommissionRate( .1 ); // set commission rate  
28  
29     System.out.printf( "\n%s:\n\n%s\n",  
30                         "Updated employee information obtained by toString", employee );  
31 } // end main  
32 } // end class CommissionEmployeeTest
```

Implicit `toString` call occurs here

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: 500.00
commission rate: 0.10

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



9.4.2 Creating and Using a BasePlusCommissionEmployee 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
5 public class BasePlusCommissionEmployee
6 {
7     private String firstName;
8     private String lastName;
9     private String socialSecurityNumber;
10    private double grossSales; // gross weekly sales
11    private double commissionRate; // commission percentage
12    private double baseSalary; // base salary per week
13
14    // six-argument constructor
15    public BasePlusCommissionEmployee( String first, String last,
16        String ssn, double sales, double rate, double salary )
17    {
18        // implicit call to Object constructor occurs here
19        firstName = first;
20        lastName = last;
21        socialSecurityNumber = ssn;
22        setGrossSales( sales ); // validate and store gross sales
```

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



```
23     setCommissionRate( rate ); // validate and store commission rate
24     setBaseSalary( salary ); // validate and store base salary
25 } // end six-argument BasePlusCommissionEmployee constructor
26
27 // set first name
28 public void setFirstName( String first )
29 {
30     firstName = first; // should validate
31 } // end method setFirstName
32
33 // return first name
34 public String getFirstName()
35 {
36     return firstName;
37 } // end method getFirstName
38
39 // set last name
40 public void setLastName( String last )
41 {
42     lastName = last; // should validate
43 } // end method setLastName
44
```

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



```
45 // return last name
46 public String getLastname()
47 {
48     return lastName;
49 } // end method getLastname
50
51 // set social security number
52 public void setSocialSecurityNumber( String ssn )
53 {
54     socialSecurityNumber = ssn; // should validate
55 } // end method setSocialSecurityNumber
56
57 // return social security number
58 public String getSocialSecurityNumber()
59 {
60     return socialSecurityNumber;
61 } // end method getSocialSecurityNumber
62
```

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



```
63     // set gross sales amount
64     public void setGrossSales( double sales )
65     {
66         if ( sales >= 0.0 )
67             grossSales = sales;
68         else
69             throw new IllegalArgumentException(
70                 "Gross sales must be >= 0.0" );
71     } // end method setGrossSales
72
73     // return gross sales amount
74     public double getGrossSales()
75     {
76         return grossSales;
77     } // end method getGrossSales
78
```

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

```
79 // set commission rate
80 public void setCommissionRate( double rate )
81 {
82     if ( rate > 0.0 && rate < 1.0 )
83         commissionRate = rate;
84     else
85         throw new IllegalArgumentException(
86             "Commission rate must be > 0.0 and < 1.0" );
87 } // end method setCommissionRate
88
89 // return commission rate
90 public double getCommissionRate()
91 {
92     return commissionRate;
93 } // end method getCommissionRate
94
```

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



```
95 // set base salary
96 public void setBaseSalary( double salary )
97 {
98     if ( salary >= 0.0 )
99         baseSalary = salary;
100    else
101        throw new IllegalArgumentException(
102            "Base salary must be >= 0.0" );
103    } // end method setBaseSalary
104
105 // return base salary
106 public double getBaseSalary()
107 {
108     return baseSalary;
109 } // end method getBaseSalary
110
111 // calculate earnings
112 public double earnings()
113 {
114     return baseSalary + ( commissionRate * grossSales );
115 } // end method earnings
116
```

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



```
117 // return String representation of BasePlusCommissionEmployee
118 @Override // indicates that this method overrides a superclass method
119 public String toString()
120 {
121     return String.format(
122         "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
123         "base-salaried commission employee", firstName, lastName,
124         "social security number", socialSecurityNumber,
125         "gross sales", grossSales, "commission rate", commissionRate,
126         "base salary", baseSalary );
127 } // end method toString
128 } // end class BasePlusCommissionEmployee
```

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 BasePlusCommissionEmployee 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     {
7         // instantiate BasePlusCommissionEmployee object
8         BasePlusCommissionEmployee employee =
9             new BasePlusCommissionEmployee(
10                "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
11
12
13        // get base-salaried commission employee data
14        System.out.println(
15            "Employee information obtained by get methods: \n" );
16        System.out.printf( "%s %s\n", "First name is",
17            employee.getFirstName() );
18        System.out.printf( "%s %s\n", "Last name is",
19            employee.getLastName() );
20        System.out.printf( "%s %s\n", "Social security number is",
21            employee.getSocialSecurityNumber() );
22        System.out.printf( "%s %.2f\n", "Gross sales is",
23            employee.getGrossSales() );
```

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



```
24     System.out.printf( "%s %.2f\n", "Commission rate is",
25         employee.getCommissionRate() );
26     System.out.printf( "%s %.2f\n", "Base salary is",
27         employee.getBaseSalary() );
28
29     employee.setBaseSalary( 1000 ); // set base salary
30
31     System.out.printf( "\n%s:\n\n%s\n",
32         "Updated employee information obtained by toString",
33         employee.toString() );
34 } // end main
35 } // end class BasePlusCommissionEmployeeTest
```

Fig. 9.7 | BasePlusCommissionEmployee test program. (Part 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.)



```
1 // Fig. 9.8: BasePlusCommissionEmployee.java
2 // private superclass members cannot be accessed in a subclass.
3
4 public class BasePlusCommissionEmployee extends CommissionEmployee
5 {
6     private double baseSalary; // base salary per week
7
8     // six-argument constructor
9     public BasePlusCommissionEmployee( String first, String last,
10         String ssn, double sales, double rate, double salary )
11    {
12        // explicit call to superclass CommissionEmployee constructor
13        super( first, last, ssn, sales, rate );
14
15        setBaseSalary( salary ); // validate and store base salary
16    } // end six-argument BasePlusCommissionEmployee constructor
17
```

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



```
18 // set base salary
19 public void setBaseSalary( double salary )
20 {
21     if ( salary >= 0.0 )
22         baseSalary = salary;
23     else
24         throw new IllegalArgumentException(
25             "Base salary must be >= 0.0" );
26 } // end method setBaseSalary
27
28 // return base salary
29 public double getBaseSalary()
30 {
31     return baseSalary;
32 } // end method getBaseSalary
33
34 // calculate earnings
35 @Override // indicates that this method overrides a superclass method
36 public double earnings()
37 {
38     // not allowed: commissionRate and grossSales private in superclass
39     return baseSalary + ( commissionRate * grossSales );
40 } // end method earnings
```

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



```
41
42     // return String representation of BasePlusCommissionEmployee
43     @Override // indicates that this method overrides a superclass method
44     public String toString()
45     {
46         // not allowed: attempts to access private superclass members
47         return String.format(
48             "%s: %s %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
49             "base-salaried commission employee", firstName, lastName,
50             "social security number", socialSecurityNumber,
51             "gross sales", grossSales, "commission rate", commissionRate,
52             "base salary", baseSalary );
53     } // end method toString
54 } // end class BasePlusCommissionEmployee
```

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



```
BasePlusCommissionEmployee.java:39: commissionRate has private access in  
CommissionEmployee  
    return baseSalary + ( commissionRate * grossSales );  
                           ^  
BasePlusCommissionEmployee.java:39: grossSales has private access in  
CommissionEmployee  
    return baseSalary + ( commissionRate * grossSales );  
                           ^  
BasePlusCommissionEmployee.java:49: firstName has private access in  
CommissionEmployee  
    "base-salaried commission employee", firstName, lastName,  
                           ^  
BasePlusCommissionEmployee.java:49: lastName has private access in  
CommissionEmployee  
    "base-salaried commission employee", firstName, lastName,  
                           ^  
BasePlusCommissionEmployee.java:50: socialSecurityNumber has private access  
in CommissionEmployee  
    "social security number", socialSecurityNumber,  
                           ^  
BasePlusCommissionEmployee.java:51: grossSales has private access in  
CommissionEmployee  
    "gross sales", grossSales, "commission rate", commissionRate,  
                           ^
```

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



```
BasePlusCommissionEmployee.java:51: commissionRate has private access in  
CommissionEmployee  
    "gross sales", grossSales, "commission rate", commissionRate,  
                           ^  
7 errors
```

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



9.4.3 Creating a CommissionEmployee-BasePlusCommission Employee Inheritance Hierarchy (Cont.)

- ▶ You can explicitly use `super()` to call the superclass's no-argument or default constructor, but this is rarely done.

```
1 // Fig. 9.9: BasePlusCommissionEmployee.java
2 // BasePlusCommissionEmployee inherits protected instance
3 // variables from CommissionEmployee.
4
5 public class BasePlusCommissionEmployee extends CommissionEmployee
6 {
7     private double baseSalary; // base salary per week
8
9     // six-argument constructor
10    public BasePlusCommissionEmployee( String first, String last,
11        String ssn, double sales, double rate, double salary )
12    {
13        super( first, last, ssn, sales, rate );
14        setBaseSalary( salary ); // validate and store base salary
15    } // end six-argument BasePlusCommissionEmployee constructor
16
```

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



```
17 // set base salary
18 public void setBaseSalary( double salary )
19 {
20     if ( salary >= 0.0 )
21         baseSalary = salary;
22     else
23         throw new IllegalArgumentException(
24             "Base salary must be >= 0.0" );
25 } // end method setBaseSalary
26
27 // return base salary
28 public double getBaseSalary()
29 {
30     return baseSalary;
31 } // end method getBaseSalary
32
33 // calculate earnings
34 @Override // indicates that this method overrides a superclass method
35 public double earnings()
36 {
37     return baseSalary + ( commissionRate * grossSales );
38 } // end method earnings
```

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



```
39
40 // return String representation of BasePlusCommissionEmployee
41 @Override // indicates that this method overrides a superclass method
42 public String toString()
43 {
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 } // end method toString
51 } // end class BasePlusCommissionEmployee
```

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

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



- ▶ Inheriting **protected** instance variables slightly **increases performance**, because we can directly access the variables in the subclass without incurring the overhead of a *set or get method call*.
- ▶ In most cases, it's better to use **private** instance variables to encourage proper software engineering, and leave code optimization issues to the compiler.
 - Code will be easier to maintain, modify and debug.



Software Engineering Observation 9.4

*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 9.5

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



Error-Prevention Tip 9.2

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

9.4.5 CommissionEmployee-BasePlusCommissionEmployee Inheritance Hierarchy Using private Instance Variables

- ▶ Hierarchy reengineered using good software engineering practices.
- ▶ Class **CommissionEmployee** declares instance variables **firstName**, **lastName**, **socialSecurityNumber**, **grossSales** and **commissionRate** as **private** and provides **public** methods for manipulating these values.





```
1 // Fig. 9.10: CommissionEmployee.java
2 // CommissionEmployee class uses methods to manipulate its
3 // private instance variables.
4 public class CommissionEmployee
5 {
6     private String firstName;
7     private String lastName;
8     private String socialSecurityNumber;
9     private double grossSales; // gross weekly sales
10    private double commissionRate; // commission percentage
11
12    // five-argument constructor
13    public CommissionEmployee( String first, String last, String ssn,
14        double sales, double rate )
15    {
16        // implicit call to Object constructor occurs here
17        firstName = first;
18        lastName = last;
19        socialSecurityNumber = ssn;
20        setGrossSales( sales ); // validate and store gross sales
21        setCommissionRate( rate ); // validate and store commission rate
22    } // end five-argument CommissionEmployee constructor
```

Fig. 9.10 | CommissionEmployee class uses methods to manipulate its **private** instance variables. (Part I of 6.)



```
23
24     // set first name
25     public void setFirstName( String first )
26     {
27         firstName = first; // should validate
28     } // end method setFirstName
29
30     // return first name
31     public String getFirstName()
32     {
33         return firstName;
34     } // end method getFirstName
35
36     // set last name
37     public void setLastName( String last )
38     {
39         lastName = last; // should validate
40     } // end method setLastName
41
```

Fig. 9.10 | CommissionEmployee class uses methods to manipulate its **private** instance variables. (Part 2 of 6.)



```
42     // return last name
43     public String getLastname()
44     {
45         return lastName;
46     } // end method getLastname
47
48     // set social security number
49     public void setSocialSecurityNumber( String ssn )
50     {
51         socialSecurityNumber = ssn; // should validate
52     } // end method setSocialSecurityNumber
53
54     // return social security number
55     public String getSocialSecurityNumber()
56     {
57         return socialSecurityNumber;
58     } // end method getSocialSecurityNumber
59
```

Fig. 9.10 | CommissionEmployee class uses methods to manipulate its **private** instance variables. (Part 3 of 6.)

```
60 // set gross sales amount
61 public void setGrossSales( double sales )
62 {
63     if ( sales >= 0.0 )
64         grossSales = sales;
65     else
66         throw new IllegalArgumentException(
67             "Gross sales must be >= 0.0" );
68 } // end method setGrossSales
69
70 // return gross sales amount
71 public double getGrossSales()
72 {
73     return grossSales;
74 } // end method getGrossSales
75
```

Fig. 9.10 | CommissionEmployee class uses methods to manipulate its **private** instance variables. (Part 4 of 6.)



```
76 // set commission rate
77 public void setCommissionRate( double rate )
78 {
79     if ( rate > 0.0 && rate < 1.0 )
80         commissionRate = rate;
81     else
82         throw new IllegalArgumentException(
83             "Commission rate must be > 0.0 and < 1.0" );
84 } // end method setCommissionRate
85
86 // return commission rate
87 public double getCommissionRate()
88 {
89     return commissionRate;
90 } // end method getCommissionRate
91
92 // calculate earnings
93 public double earnings()
94 {
95     return getCommissionRate() * getGrossSales();
96 } // end method earnings
97
```

Fig. 9.10 | CommissionEmployee class uses methods to manipulate its **private** instance variables. (Part 5 of 6.)



```
98     // return String representation of CommissionEmployee object
99     @Override // indicates that this method overrides a superclass method
100    public String toString()
101    {
102        return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
103                           "commission employee", getFirstName(), getLastName(),
104                           "social security number", getSocialSecurityNumber(),
105                           "gross sales", getGrossSales(),
106                           "commission rate", getCommissionRate() );
107    } // end method toString
108 } // end class CommissionEmployee
```

Fig. 9.10 | CommissionEmployee class uses methods to manipulate its **private** instance variables. (Part 6 of 6.)



```
1 // Fig. 9.11: BasePlusCommissionEmployee.java
2 // BasePlusCommissionEmployee class inherits from CommissionEmployee
3 // and accesses the superclass's private data via inherited
4 // public methods.
5
6 public class BasePlusCommissionEmployee extends CommissionEmployee
7 {
8     private double baseSalary; // base salary per week
9
10    // six-argument constructor
11    public BasePlusCommissionEmployee( String first, String last,
12        String ssn, double sales, double rate, double salary )
13    {
14        super( first, last, ssn, sales, rate );
15        setBaseSalary( salary ); // validate and store base salary
16    } // end six-argument BasePlusCommissionEmployee constructor
17
```

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



```
18 // set base salary
19 public void setBaseSalary( double salary )
20 {
21     if ( salary >= 0.0 )
22         baseSalary = salary;
23     else
24         throw new IllegalArgumentException(
25             "Base salary must be >= 0.0" );
26 } // end method setBaseSalary
27
28 // return base salary
29 public double getBaseSalary()
30 {
31     return baseSalary;
32 } // end method getBaseSalary
33
34 // calculate earnings
35 @Override // indicates that this method overrides a superclass method
36 public double earnings()
37 {
38     return getBaseSalary() + super.earnings();
39 } // end method earnings
```

Fig. 9.11 | BasePlusCommissionEmployee class inherits from CommissionEmployee and accesses the superclass's `private` data via inherited

methods (Fig. 9.12).



```
40 // return String representation of BasePlusCommissionEmployee
41 @Override // indicates that this method overrides a superclass method
42 public String toString()
43 {
44     return String.format( "%s %s\n%s: %.2f", "base-salaried",
45                           super.toString(), "base salary", getBaseSalary() );
46 }
47 } // end method toString
48 } // end class BasePlusCommissionEmployee
```

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



Common Programming Error 9.3

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 a dot (.) separator when calling the superclass's method causes the subclass method to call itself, potentially creating an error called infinite recursion. Recursion, used correctly, is a powerful capability discussed in Chapter 18.



Software Engineering Observation 9.6

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 booleans, null for references).



9.5 Object Class

- ▶ All classes in Java inherit directly or indirectly from `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
<code>clone</code>	This protected method, which takes no arguments and returns an <code>Object</code> 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 <code>clone</code> method's implementation would perform a deep copy that creates a new object for each reference-type instance variable. Implementing <code>clone</code> correctly is difficult. For this reason, its use is discouraged. Many industry experts suggest that object serialization should be used instead. We discuss object serialization in Chapter 17, Files, Streams and Object Serialization.

Fig. 9.12 | Object methods. (Part 1 of 3.)



Method	Description
equals	This method compares two objects for equality and returns <code>true</code> if they're equal and <code>false</code> otherwise. The method takes any <code>Object</code> as an argument. When objects of a particular class must be compared for equality, the class should override method <code>equals</code> to compare the <i>contents</i> of the two objects. For the requirements of implementing this method, refer to the method's documentation at download.oracle.com/javase/6/docs/api/java/lang/Object.html#equals(java.lang.Object) . The default <code>equals</code> implementation uses operator <code>==</code> to determine whether two references <i>refer to the same object</i> in memory. Section 16.3.3 demonstrates class <code>String</code> 's <code>equals</code> method and differentiates between comparing <code>String</code> objects with <code>==</code> and with <code>equals</code> .
finalize	This <code>protected</code> method (introduced in Section 8.10) is called by the garbage collector to perform termination housekeeping on an object just before the garbage collector reclaims the object's memory. Recall that it's unclear whether, or when, method <code>finalize</code> will be called. For this reason, most programmers should avoid method <code>finalize</code> .

Fig. 9.12 | Object methods (Part 2 of 3.)



Method	Description
getClass	Every object in Java knows its own type at execution time. Method <code>getClass</code> (used in Sections 10.5, 14.5 and 24.3) returns an object of class <code>Class</code> (package <code>java.lang</code>) that contains information about the object's type, such as its class name (returned by <code>Class</code> method <code>getName</code>).
hashCode	Hashcodes are <code>int</code> values that are useful for high-speed storage and retrieval of information stored in a data structure that's known as a hashtable (discussed in Section 20.11). This method is also called as part of class <code>Object</code> 's default <code>toString</code> method implementation.
wait, notify, notifyAll	Methods <code>notify</code> , <code>notifyAll</code> and the three overloaded versions of <code>wait</code> are related to multithreading, which is discussed in Chapter 26.
<code>toString</code>	This method (introduced in Section 9.4.1) returns a <code>String</code> representation of an object. The default implementation of this method returns the package name and class name of the object's class followed by a hexadecimal representation of the value returned by the object's <code>hashCode</code> method.

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



9.6 Designing with Composition vs. inheritance

▶ Inheritance-Based Designs

- Inheritance creates **tight coupling** among the classes in a hierarchy
- Inheritance is done at compile time.
- you should use inheritance only for **true is-a** relationships in which you can assign a subclass object to a superclass reference.



9.6 Designing with Composition vs. inheritance(Cont.)

▶ Composition-Based Designs

- Composition is loosely coupled.
- Composition can be done at compile time(final reference), but it also can be done at execution time(non-final referece).



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.