Chapter 9 Object-Oriented Programming: Inheritance

Java™ How to Program, 9/e

OBJECTIVES

In this chapter you'll learn:

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

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.
  - 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
- 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.

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.
- 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.

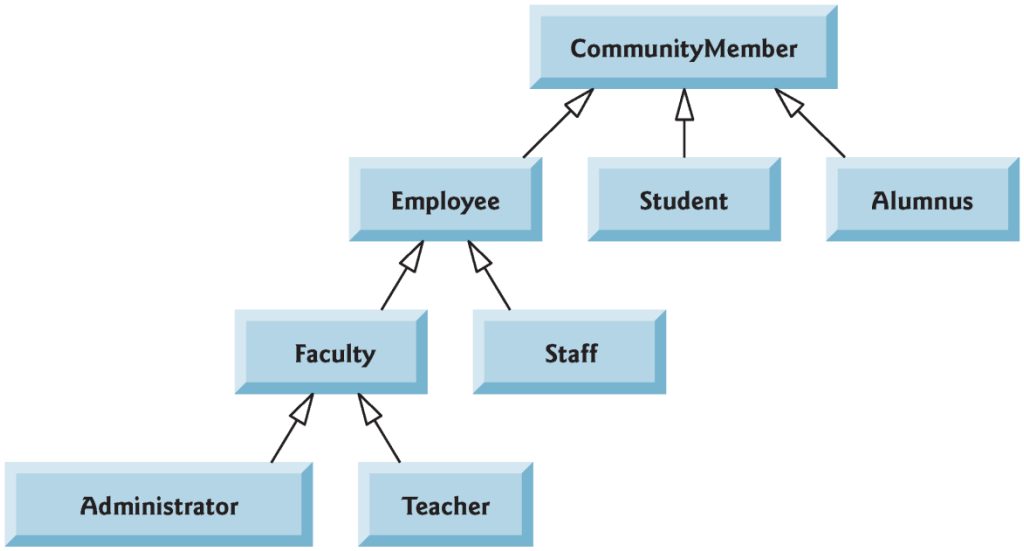

Fig. 9.2 | Inheritance hierarchy for university `CommunityMembers`.

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`.

9.3 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 in 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 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  {
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 1 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.)

- 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.

A sub-class can always widen the access modifier, because it is still a valid substitution for the super-class.

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.*

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 lines 6–10 of Fig. 9.4 as follows:

```
protected String firstName;
protected String lastName;
protected 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 (51 lines) is considerably shorter than that in Fig. 9.6 (128 lines)
  - 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 commission employee exists only in class `CommissionEmployee`.

9.4.4 CommissionEmployee–BasePlus-CommissionEmployee 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.

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 software 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.

9.5 Constructors in Subclasses

- Instantiating a subclass object begins a chain of constructor calls
  - The subclass constructor, before performing its own tasks, invokes its direct superclass's 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 class `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.

Super keyword

- Usage of Java super Keyword

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

Example

```
1.  class Animal{
2.  String color="white";
3.  }
4.  class Dog extends Animal{
5.  String color="black";
6.  Dog(){
7.  Super(); //calls super class constructor
8.  }
9.  void printColor(){
10. System.out.println(color);//prints color of Dog class
11. System.out.println(super.color);//prints color of Animal class
12. }
13. }
14. class TestSuper1{
15. public static void main(String args[]){
16. Dog d=new Dog();
17. d.printColor();
18. }}
```