Chapter 9 Object-Oriented Programming: Inheritance

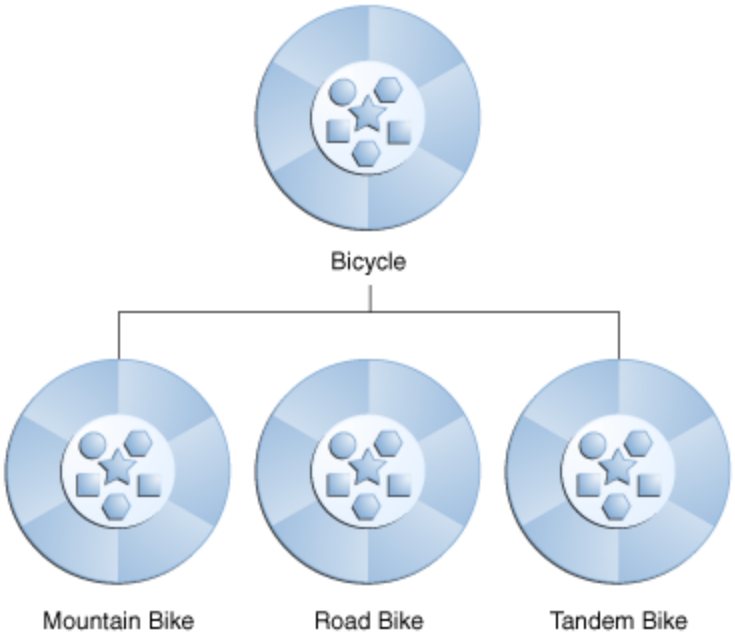
Java™ How to Program, 8/e

What is Inheritance?

- Different kinds of objects often have a certain amount in common with each other.
- Mountain bikes, road bikes, and tandem bikes, for example, all share the characteristics of bicycles (current speed, current pedal cadence, current gear).
- each also defines additional features that make them different:
  - tandem bicycles have two seats and two sets of handlebars;
  - road bikes have drop handlebars;
  - some mountain bikes have an additional chain ring, giving them a lower gear ratio.
- Object-oriented programming allows classes to *inherit* commonly used state and behavior from other classes.

What is Inheritance?

- In this example, Bicycle now becomes the *superclass* ofMountainBike, RoadBike, and TandemBike.
- In the Java programming language;
  - each class is allowed to have one direct superclass
  - each superclass has the potential for an unlimited number of *subclasses*:



Inheritance Overview

- **Ideas**
- You can make a class that "inherits" characteristics of another class
  - The original class is called **"parent class", "super class", or "base class".**
  - The new class is called **"child class", "subclass", or "extended class".**
- The child class has access to all non-private methods of the parent class.
  - **No special syntax need to call inherited methods**
- **Syntax**

```
public class ChildClass extends ParentClass { … }
```

Why Inheritance?

- Supports the key OOP idea of **code reuse** (i.e., **don't write the same code twice**).
  - inheritance can greatly help reduce code redundancy between similar objects by **taking what those objects have in common and putting them in one place→***put common code in superclass*
  - This also creates more maintainable code

- Design class hierarchies so that shared behavior is in inherited to all classes that need it.

Simple Example

- **Person**

```
public class Person {
  public String getFirstName() { … }
  public String getLastName() { … }
}
```

- **Employee**

```
public class Employee extends Person {
  public double getSalary() { … }

  public String getEmployeeInfo() {
    return(getFirstName() + " " + getLastName() +
           " earns " + getSalary());
  }
}
```

Example from Eclipse-Ship

Important Points

- Format for defining subclasses

public class Speedboat extends Ship {....

  - The old class is called the **superclass, base class or parent class**
  - The new class is called the **subclass, derived class or child class**

- Using inherited methods

setColor(color);

- Using super(...) for inherited constructors
  - *Only when the zero-arg super constructor is not OK*
  - **super(x, y, speed, direction, name);**
- Using super.someMethod(...) for inherited methods
  - *Only when there is a name conflict*
  - **super.printLocation();**

Inheritance

- **Effect of inheritance**
  - Subclasses automatically have all public fields and methods of the parent class
    - don't need any special syntax to access the inherited fields and methods
    - use exact same syntax as with locally defined fields or methods.
  - can also add in fields or methods not available in the superclass
- **Java doesn't support multiple inheritance**
  - A class can only have one *direct parent. But grandparent and greatgrandparent* (etc.) are legal and common.

Inherited constructors and super(...)

- **When you instantiate an object of a subclass, the system will automatically call the superclass constructor first**
  - By default, the zero-argument superclass constructor is called
  - If you want to specify that a different parent constructor is called, invoke the parent class constructor with super(*args)*
  - If super(…) is used in a subclass constructor, then super(…) must be the first statement in the constructor

Inherited constructors and super(...)

- **Constructor life-cycle**
  - Each constructor has three phases:
    1. Invoke the constructor of the superclass
       - zero-argument constructor is called automatically.
       - No special syntax is needed unless you want a *different parent constructor.*
    2. Initialize all instance variables based on their initialization statements
    3. Execute the body of the constructor

Overridden methods and super.method(...)

- When a class defines a method using the same name, return type, and arguments as a method in the superclass, then the class *overrides the method in the superclass*
  - Only non-static methods can be overridden
  - If there is a locally defined method and an inherited method that have the same name and take the same arguments, you can use the following to refer to the inherited method super.methodName(...)
  - Successive use of super (super.super.methodName) is not legal.

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`.

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

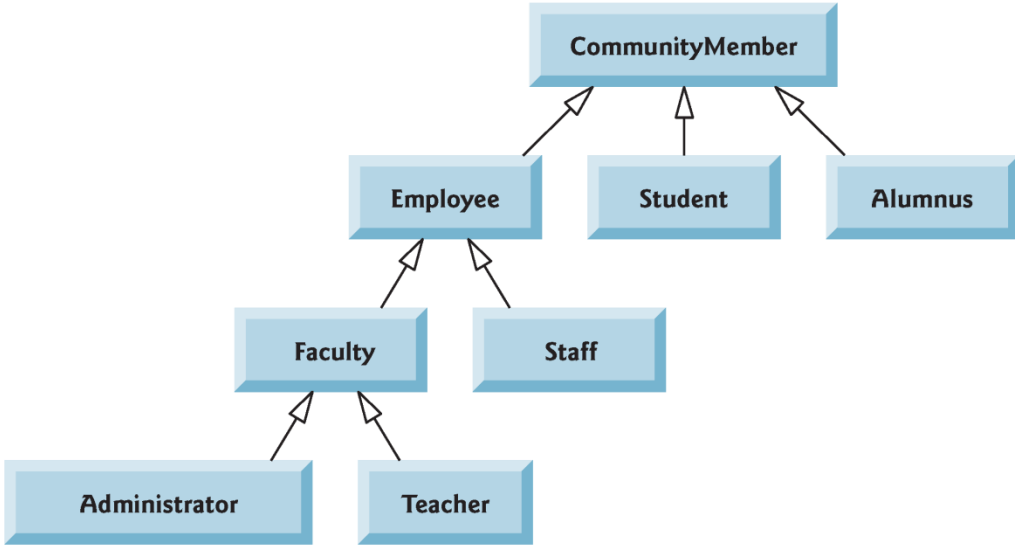
- Figure 9.1 lists several simple examples of superclasses and subclasses
  - Superclasses tend to be “more general” and subclasses “more specific.”
- Because every subclass object *is an* object of its superclass, and one superclass can have many subclasses, the set of objects represented by a superclass 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.
- 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.

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`.
  - In the first hw, student has an advisor.

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.

9.4 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.

9.5 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.

9.5.1 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
```

`extends Object` not required; this will be done implicitly

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

```
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 5.)

```
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
60      // set gross sales amount
61      public void setGrossSales( double sales )
62      {
63         grossSales = ( sales < 0.0 ) ? 0.0 : sales;
64      } // end method setGrossSales
```

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

```
65
66      // return gross sales amount
67      public double getGrossSales()
68      {
69         return grossSales;
70      } // end method getGrossSales
71
72      // set commission rate
73      public void setCommissionRate( double rate )
74      {
75         commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
76      } // end method setCommissionRate
77
78      // return commission rate
79      public double getCommissionRate()
80      {
81         return commissionRate;
82      } // end method getCommissionRate
83
```

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

```
84      // calculate earnings
85      public double earnings()
86      {
87         return commissionRate * grossSales;
88      } // end method earnings
89
90      // return String representation of CommissionEmployee object
91      @Override // indicates that this method overrides a superclass method
92      public String toString()
93      {
94         return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
95            "commission employee", firstName, lastName,
96            "social security number", socialSecurityNumber,
97            "gross sales", grossSales,
98            "commission rate", commissionRate );
99      } // end method toString
100  } // end class CommissionEmployee
```

Overridden `toString` customizes how this method works for a `CommissionEmploye`; `@Override` helps compiler ensure that the method has the same signature as a method in the superclass

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

9.5.1 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.

9.5.1 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.

9.5.1 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  {
6     public static void main( String[] args )
7     {
8        // instantiate CommissionEmployee object
9        CommissionEmployee employee = new CommissionEmployee(
10          "Sue", "Jones", "222-22-2222", 10000, .06 );
11
12       // get commission employee data
13       System.out.println(
14          "Employee information obtained by get methods: \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.)

9.5.2 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 that 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
```

The only new piece of data in class `BasePlusCommissionEmployee`

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 1 of 6.)

```
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
```

Initializes the base salary

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 2 of 6.)

```
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
63      // set gross sales amount
64      public void setGrossSales( double sales )
65      {
66         grossSales = ( sales < 0.0 ) ? 0.0 : sales;
67      } // end method setGrossSales
```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 3 of 6.)

```
68
69      // return gross sales amount
70      public double getGrossSales()
71      {
72         return grossSales;
73      } // end method getGrossSales
74
75      // set commission rate
76      public void setCommissionRate( double rate )
77      {
78         commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
79      } // end method setCommissionRate
80
81      // return commission rate
82      public double getCommissionRate()
83      {
84         return commissionRate;
85      } // end method getCommissionRate
86
```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 4 of 6.)

```
87      // set base salary
88      public void setBaseSalary( double salary )
89      {
90         baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
91      } // end method setBaseSalary
92
93      // return base salary
94      public double getBaseSalary()
95      {
96         return baseSalary;
97      } // end method getBaseSalary
98
99      // calculate earnings
100     public double earnings()
101     {
102        return baseSalary + ( commissionRate * grossSales );
103     } // end method earnings
104
```

Similar to `CommissionEmployee`'s `earnings` method

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 5 of 6.)

```
105    // return String representation of BasePlusCommissionEmployee
106    @Override // indicates that this method overrides a superclass method
107    public String toString()
108    {
109       return String.format(
110          "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f\n%s: %.2f",
111          "base-salaried commission employee", firstName, lastName,
112          "social security number", socialSecurityNumber,
113          "gross sales", grossSales, "commission rate", commissionRate,
114          "base salary", baseSalary );
115    } // end method toString
116 } // end class BasePlusCommissionEmployee
```

Similar to `CommissionEmployee`'s `toString` method

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

9.5.2 Creating and Using a BasePlusCommissionEmployee 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 nearly identical
  - `BasePlusCommissionEmployee`'s `toString` also outputs instance variable `baseSalary`

9.5.2 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 nightmare.

Software Engineering Observation 9.3

With inheritance, the common instance variables and methods of all the classes in the hierarchy are declared in a superclass. When changes are made for these common features in the superclass—subclasses then inherit the changes. Without inheritance, changes would need to be made to all the source-code files that contain a copy of the code in question.

9.5.3 Creating a CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy

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

```
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
18    // set base salary
19    public void setBaseSalary( double salary )
20    {
21       baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
22    } // end method setBaseSalary
```

New subclass of `CommissionEmployee`

Must call superclass constructor first

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

```
23
24      // return base salary
25      public double getBaseSalary()
26      {
27         return baseSalary;
28      } // end method getBaseSalary
29
30      // calculate earnings
31      @Override // indicates that this method overrides a superclass method
32      public double earnings()
33      {
34         // not allowed: commissionRate and grossSales private in superclass
35         return baseSalary + ( commissionRate * grossSales );
36      } // end method earnings
37
```

CommissionEmployee `private` instance variables are not accessible here

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

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

CommissionEmployee `private` instance variables are not accessible here

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

```
BasePlusCommissionEmployee.java:35: commissionRate has private access in
CommissionEmployee
      return baseSalary + ( commissionRate * grossSales );
                            ^
BasePlusCommissionEmployee.java:35: grossSales has private access in
CommissionEmployee
      return baseSalary + ( commissionRate * grossSales );
                                             ^
BasePlusCommissionEmployee.java:45: firstName has private access in
CommissionEmployee
        "base-salaried commission employee", firstName, lastName,
                                             ^
BasePlusCommissionEmployee.java:45: 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:46: socialSecurityNumber has private access
in CommissionEmployee
         "social security number", socialSecurityNumber,
                                   ^
BasePlusCommissionEmployee.java:47: grossSales has private access in
CommissionEmployee
         "gross sales", grossSales, "commission rate", commissionRate,
                        ^
BasePlusCommissionEmployee.java:47: 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.)

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

- Each subclass constructor must implicitly or explicitly call its superclass constructor 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 subclass constructor's body.
- If the subclass constructor did not invoke the superclass's constructor explicitly, Java would attempt to invoke the superclass's no-argument or default 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.

9.5.3 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.

9.5.4 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 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.

9.5.4 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 (47 lines) is considerably shorter than that in Fig. 9.6 (116 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`.

```
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
17    // set base salary
18    public void setBaseSalary( double salary )
19    {
20       baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
21    } // end method setBaseSalary
22
```

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

```
23      // return base salary
24      public double getBaseSalary()
25      {
26         return baseSalary;
27      } // end method getBaseSalary
28
29      // calculate earnings
30      @Override // indicates that this method overrides a superclass method
31      public double earnings()
32      {
33         return baseSalary + ( commissionRate * grossSales );
34      } // end method earnings
35
```

CommissionEmployee `protected` instance variables *are* accessible here

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

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

CommissionEmployee protected instance variables *are* accessible here

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

9.5.4 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.

9.5.4 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 problematic, 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 9.4

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.*

9.5.5 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  {
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
23
```

Data is `private` for best encapsulation; makes code easier to maintain/debug.

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

```
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
42      // return last name
43      public String getLastName()
44      {
45         return lastName;
46      } // end method getLastName
```

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

```
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
60      // set gross sales amount
61      public void setGrossSales( double sales )
62      {
63         grossSales = ( sales < 0.0 ) ? 0.0 : sales;
64      } // end method setGrossSales
65
66      // return gross sales amount
67      public double getGrossSales()
68      {
```

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

```
69          return grossSales;
70       } // end method getGrossSales
71
72       // set commission rate
73       public void setCommissionRate( double rate )
74       {
75          commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
76       } // end method setCommissionRate
77
78       // return commission rate
79       public double getCommissionRate()
80       {
81          return commissionRate;
82       } // end method getCommissionRate
83
84       // calculate earnings
85       public double earnings()
86       {
87          return getCommissionRate() * getGrossSales();
88       } // end method earnings
89
```

No longer accessing instance variables directly here

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

```
90      // return String representation of CommissionEmployee object
91      @Override // indicates that this method overrides a superclass method
92      public String toString()
93      {
94         return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
95            "commission employee", getFirstName(), getLastName(),
96            "social security number", getSocialSecurityNumber(),
97            "gross sales", getGrossSales(),
98            "commission rate", getCommissionRate() );
99      } // end method toString
100  } // end class CommissionEmployee
```

No longer accessing instance variables directly here

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

9.5.5 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
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
18    // set base salary
19    public void setBaseSalary( double salary )
20    {
21       baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
22    } // end method setBaseSalary
```

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

```
23
24      // return base salary
25      public double getBaseSalary()
26      {
27         return baseSalary;
28      } // end method getBaseSalary
29
30      // calculate earnings
31      @Override // indicates that this method overrides a superclass method
32      public double earnings()
33      {
34         return getBaseSalary() + super.earnings();
35      } // end method earnings
36
37      // return String representation of BasePlusCommissionEmployee
38      @Override // indicates that this method overrides a superclass method
39      public String toString()
40      {
41         return String.format( "%s %s\n%s: %.2f", "base-salaried",
42            super.toString(), "base salary", getBaseSalary() );
43      } // end method toString
44   } // end class BasePlusCommissionEmployee
```

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

9.5.5 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 9.4

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, Recursion.*

9.6 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.

Software Engineering Observation 9.6

When a program creates a subclass object, the subclass constructor immediately calls the superclass constructor (explicitly, via `super`*, or implicitly). The superclass constructor's body executes to initialize the superclass's instance variables that are part of the subclass object, then the subclass constructor's body executes to initialize the subclass-only instance variables. 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).*