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.

```
21          System.out.printf(
22             "Date object constructor for date %s\n", this );
23       } // end Date constructor
24
25       // utility method to confirm proper month value
26       private int checkMonth( int testMonth )
27       {
28          if ( testMonth > 0 && testMonth <= 12 ) // validate month
29             return testMonth;
30          else // month is invalid
31             throw new IllegalArgumentException( "month must be 1-12" );
32       } // end method checkMonth
33
34       // utility method to confirm proper day value based on month and year
35       private int checkDay( int testDay )
36       {
37          // check if day in range for month
38          if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
39             return testDay;
40
```

Fig. 8.7 | `Date` class declaration. (Part 2 of 3.)

```
41         // check for leap year
42         if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
43              ( year % 4 == 0 && year % 100 != 0 ) ) )
44            return testDay;
45
46         throw new IllegalArgumentException(
47            "day out-of-range for the specified month and year" );
48      } // end method checkDay
49
50      // return a String of the form month/day/year
51      public String toString()
52      {
53         return String.format( "%d/%d/%d", month, day, year );
54      } // end method toString
55   } // end class Date
```

Fig. 8.7 | `Date` class declaration. (Part 3 of 3.)

```
1  // Fig. 8.8: Employee.java
2  // Employee class with references to other objects.
3
4  public class Employee
5  {
6     private String firstName;
7     private String lastName;
8     private Date birthDate;
9     private Date hireDate;
10
11    // constructor to initialize name, birth date and hire date
12    public Employee( String first, String last, Date dateOfBirth,
13       Date dateOfHire )
14    {
15       firstName = first;
16       lastName = last;
17       birthDate = dateOfBirth;
18       hireDate = dateOfHire;
19    } // end Employee constructor
20
```

Fig. 8.8 | `Employee` class with references to other objects. (Part 1 of 2.)

```
21      // convert Employee to String format
22      public String toString()
23      {
24         return String.format( "%s, %s  Hired: %s  Birthday: %s",
25            lastName, firstName, hireDate, birthDate );
26      } // end method toString
27   } // end class Employee
```

Fig. 8.8 | `Employee` class with references to other objects. (Part 2 of 2.)

```
1  // Fig. 8.9: EmployeeTest.java
2  // Composition demonstration.
3
4  public class EmployeeTest
5  {
6     public static void main( String[] args )
7     {
8        Date birth = new Date( 7, 24, 1949 );
9        Date hire = new Date( 3, 12, 1988 );
10       Employee employee = new Employee( "Bob", "Blue", birth, hire );
11
12       System.out.println( employee );
13    } // end main
14 } // end class EmployeeTest
```

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Blue, Bob  Hired: 3/12/1988  Birthday: 7/24/1949
```

Fig. 8.9 | Composition demonstration.

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.

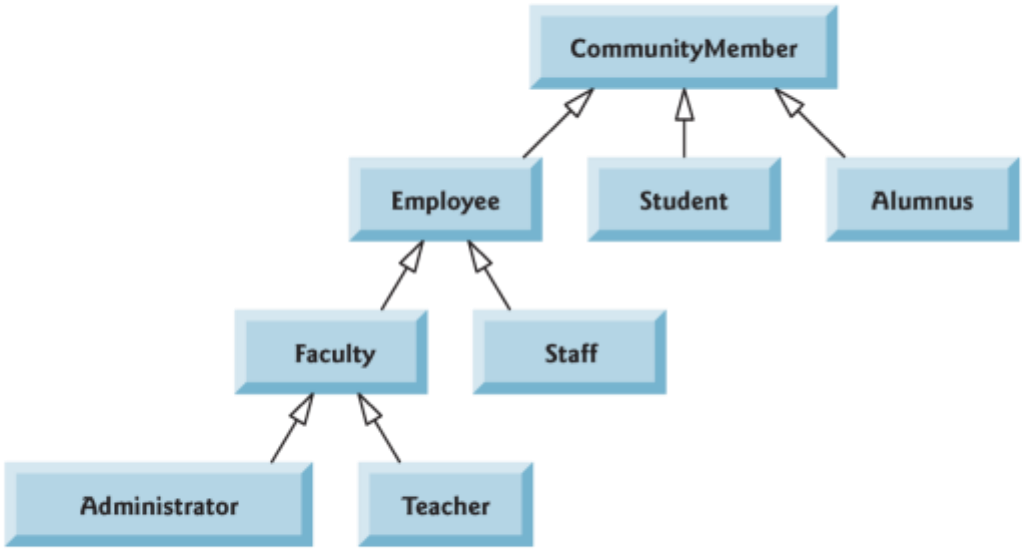

Fig. 9.2 | Inheritance hierarchy for university `CommunityMembers`.

9.2 Superclasses and Subclasses (Cont.)

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

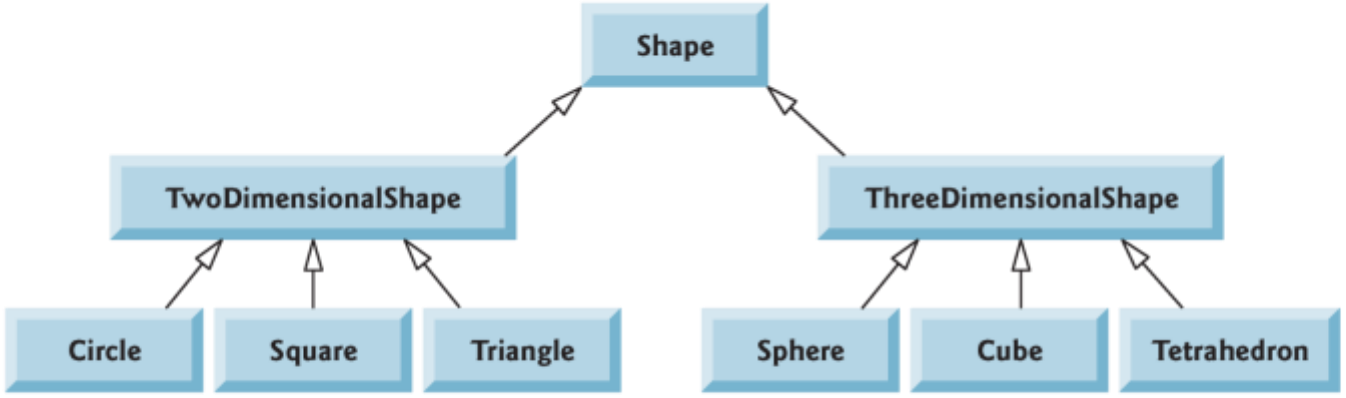

Fig. 9.3 | Inheritance hierarchy for Shapes.

9.2 Superclasses and Subclasses (Cont.)

- Not every class relationship is an inheritance relationship.
- *Has-a* relationship
  - Create classes by composition of existing classes.
  - Example: Given the classes `Employee`, `BirthDate` and `TelephoneNumber`, it's improper to say that an `Employee` *is a* `BirthDate` or that an `Employee` *is a* `TelephoneNumber`.
  - However, an `Employee` *has a* `BirthDate`, and an `Employee` *has a* `TelephoneNumber`.

9.2 Superclasses and Subclasses (Cont.)

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

9.3 protected Members

- A class's `public` members are accessible wherever the program has a reference to an object of that class or one of its subclasses.
- A class's `private` members are accessible only within the class itself.
- protected access is an intermediate level of access between `public` and `private`.
  - A superclass's `protected` members can be accessed by members of that superclass, by members of its subclasses and by members of other classes in the same package
    - `protected` members also have package access.
  - All `public` and `protected` superclass members retain their original access modifier when they become members of the subclass.

9.3 protected Members (Cont.)

- A superclass's `private` members are hidden 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.)

- `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  {
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.)

9.4.2 Creating and Using a BasePlus-CommissionEmployee Class

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

```
1  // Fig. 9.6: BasePlusCommissionEmployee.java
2  // BasePlusCommissionEmployee class represents an employee who receives
3  // a base salary in addition to commission.
4
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 1 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\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 BasePlus-CommissionEmployee Class (Cont.)

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

```
1  // Fig. 9.7: BasePlusCommissionEmployeeTest.java
2  // BasePlusCommissionEmployee test program.
3
4  public class BasePlusCommissionEmployeeTest
5  {
6     public static void main( String[] args )
7     {
8        // instantiate BasePlusCommissionEmployee object
9        BasePlusCommissionEmployee employee =
10          new BasePlusCommissionEmployee(
11          "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
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.)

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

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

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

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

9.4.3 Creating a CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy

- Class `BasePlusCommissionEmployee` class extends class `CommissionEmployee`
- A `BasePlusCommissionEmployee` object *is a* `CommissionEmployee`
  - Inheritance passes on class `CommissionEmployee`'s capabilities.
- Class `BasePlusCommissionEmployee` also has instance variable `baseSalary`.
- Subclass `BasePlusCommissionEmployee` inherits `CommissionEmployee`'s instance variables and methods
  - Only 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
```

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

```
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: %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 5.)

```
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–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.

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

- Compilation errors occur when the subclass attempts to access the superclass's `private` instance variables.
- These lines could have used appropriate *get* methods to retrieve the values of the superclass's instance variables.

9.4.4 CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy Using protected Instance Variables

- To enable a subclass to directly access superclass instance variables, we can declare those members as `protected` in the superclass.
- New `CommissionEmployee` class modified only 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`.

```
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–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.

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–BasePlus-CommissionEmployee 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.

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

- `CommissionEmployee` methods `earnings` and `toString` use the class's *get* methods to obtain the values of its instance variables.
  - If we decide to change the internal representation of the data (e.g., variable names) only the bodies of the *get and set methods that directly manipulate the instance variables will need to change.*
  - These changes occur solely within the superclass-—no changes to the subclass are needed.
  - Localizing the effects of changes like this is a good software engineering practice.
- Subclass `BasePlusCommissionEmployee` inherits `Commission-Employee`'s non-`private` methods and can access the `private` superclass members via those methods.

```
1  // Fig. 9.10: CommissionEmployee.java
2  // CommissionEmployee class uses methods to manipulate its
3  // private instance variables.
4  public class CommissionEmployee
5  {
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 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         la5stName = 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.)

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

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

```
1  // Fig. 9.11: BasePlusCommissionEmployee.java
2  // BasePlusCommissionEmployee class inherits from CommissionEmployee
3  // and accesses the superclass's private data via inherited
4  // public methods.
5
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 1 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 `public` methods. (Part 2 of 3.)

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

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

- Method `earnings` overrides class the superclass's `earnings` method.
- The new version calls `CommissionEmployee`'s `earnings` method with `super.earnings()`.
  - Obtains the earnings based on commission alone
- Placing the keyword `super` and a dot (`.`) separator before the superclass method name invokes the superclass version of an overridden method.
- Good software engineering practice
  - If a method performs all or some of the actions needed by another method, call that method rather than duplicate its code.

Common Programming Error 9.3

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.*

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

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

9.5 Constructors in Subclasses

- Instantiating a subclass object begins a chain of constructor calls
  - The subclass constructor, before performing its own tasks, 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

Java ensures that even if a constructor does not assign a value to an instance variable, the variable is still initialized to its default value (e.g., `0` for primitive numeric types, `false` for `boolean`s, `null` for references).

9.6 Software Engineering with Inheritance

- When you extend a class, the new class inherits the superclass's members—though the `private` superclass members are hidden in the new class.
- You can customize the new class to meet your needs by including additional members and by overriding superclass members.
  - Doing this does not require the subclass programmer to change (or even have access to) the superclass's source code.
  - Java simply requires access to the superclass's `.class` file.

Software Engineering Observation 9.7

Although inheriting from a class does not *require access to the class's source code, developers often insist on seeing the source code to understand how the class is implemented. Developers in industry want to ensure that they're extending a solid class—for example, a class that performs well and is implemented robustly and securely.*

Software Engineering Observation 9.8

At the design stage in an object-oriented system, you'll often find that certain classes are closely related. You should "factor out" common instance variables and methods and place them in a superclass. Then use inheritance to develop subclasses, specializing them with capabilities beyond those inherited from the superclass.

Software Engineering Observation 9.9

Declaring a subclass does not affect its superclass's source code. Inheritance preserves the integrity of the superclass.

Software Engineering Observation 9.10

Designers of object-oriented systems should avoid class proliferation. Such proliferation creates management problems and can hinder software reusability, because in a huge class library it becomes difficult to locate the most appropriate classes. The alternative is to create fewer classes that provide more substantial functionality, but such classes might prove cumbersome.

9.7 `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
`clone`	This `protected` method, which takes no arguments and returns an `Object` reference, makes a copy of the object on which it's called. The default implementation performs a so-called **shallow copy**—instance-variable values in one object are copied into another object of the same type. For reference types, only the references are copied. A typical overridden `clone` method's implementation would perform a **deep copy** that creates a new object for each reference-type instance variable. Implementing `clone` correctly is difficult. For this reason, its use is discouraged. 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 `true` if they're equal and `false` otherwise. The method takes any `Object` as an argument. When objects of a particular class must be compared for equality, the class should override method `equals` to compare the *contents* of the two objects. For the requirements of implementing this method, refer to the method's documentation at `download.oracle.com/javase/6/docs/api/java/lang/Object.html# equals(java.lang.Object)`. The default `equals` implementation uses operator == to determine whether two references *refer to the same object* in memory. Section 16.3.3 demonstrates class `String`'s `equals` method and differentiates between comparing `String` objects with == and with `equals`.
`finalize`	This `protected` 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 `finalize` will be called. For this reason, most programmers should avoid method `finalize`.

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

Method	Description
`getClass`	Every object in Java knows its own type at execution time. Method `getClass` (used in Sections 10.5, 14.5 and 24.3) returns an object of class `Class` (package `java.lang`) that contains information about the object's type, such as its class name (returned by `Class` method `getName`).
`hashCode`	Hashcodes are `int` 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 `Object`'s default `toString` method implementation.
`wait`, `notify`, `notifyAll`	Methods `notify`, `notifyAll` and the three overloaded versions of `wait` are related to multithreading, which is discussed in Chapter 26.
`toString`	This method (introduced in Section 9.4.1) returns a `String` representation of an object. The default implementation of this method returns the package name and class name of the object's class followed by a hexadecimal representation of the value returned by the object's `hashCode` method.

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

9.8 (Optional) GUI and Graphics Case Study: Displaying Text and Images Using Labels

- Labels are a convenient way of identifying GUI components on the screen and keeping the user informed about the current state of the program.
- A JLabel (from package `javax.swing`) can display text, an image or both.
- The example in Fig. 9.13 demonstrates several `JLabel` features, including a plain text label, an image label and a label with both text and an image.

```
1  // Fig 9.13: LabelDemo.java
2  // Demonstrates the use of labels.
3  import java.awt.BorderLayout;
4  import javax.swing.ImageIcon;
5  import javax.swing.JLabel;
6  import javax.swing.JFrame;
7
8  public class LabelDemo
9  {
10    public static void main( String[] args )
11    {
12       // Create a label with plain text
13       JLabel northLabel = new JLabel( "North" );
14
15       // create an icon from an image so we can put it on a JLabel
16       ImageIcon labelIcon = new ImageIcon( "GUItip.gif" );
17
18       // create a label with an Icon instead of text
19       JLabel centerLabel = new JLabel( labelIcon );
20
21       // create another label with an Icon
22       JLabel southLabel = new JLabel( labelIcon );
23
```

Fig. 9.13 | JLabel with text and with images. (Part 1 of 3.)

```
24          // set the label to display text (as well as an icon)
25          southLabel.setText( "South" );
26
27          // create a frame to hold the labels
28          JFrame application = new JFrame();
29
30          application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
31
32          // add the labels to the frame; the second argument specifies
33          // where on the frame to add the label
34          application.add( northLabel, BorderLayout.NORTH );
35          application.add( centerLabel, BorderLayout.CENTER );
36          application.add( southLabel, BorderLayout.SOUTH );
37
38          application.setSize( 300, 300 ); // set the size of the frame
39          application.setVisible( true ); // show the frame
40       } // end main
41    } // end class LabelDemo
```

Fig. 9.13 | `JLabel` with text and with images. (Part 2 of 3.)

Fig. 9.13 | `JLabel` with text and with images. (Part 3 of 3.)

9.8 (Optional) GUI and Graphics Case Study: Displaying Text and Images Using Labels (Cont.)

- An ImageIcon represents an image that can be displayed on a `JLabel`.
- The constructor for `ImageIcon` receives a `String` that specifies the path to the image.
- `ImageIcon` can load images in GIF, JPEG and PNG image formats.
- `JLabel` method setText changes the text the label displays.

9.8 (Optional) GUI and Graphics Case Study: Displaying Text and Images Using Labels (Cont.)

- An overloaded version of method `add` that takes two parameters allows you to specify the GUI component to add to a `JFrame` and the location in which to add it.
  - The first parameter is the component to attach.
  - The second is the region in which it should be placed.
- Each `JFrame` has a layout to position GUI components.
  - Default layout for a `JFrame` is BorderLayout.
  - Five regions—`NORTH` (top), `SOUTH` (bottom), `EAST` (right side), `WEST` (left side) and `CENTER` (constants in class `BorderLayout`)
  - Each region is declared as a constant in class `BorderLayout`.
- When calling method `add` with one argument, the `JFrame` places the component in the `BorderLayout`'s `CENTER` automatically.

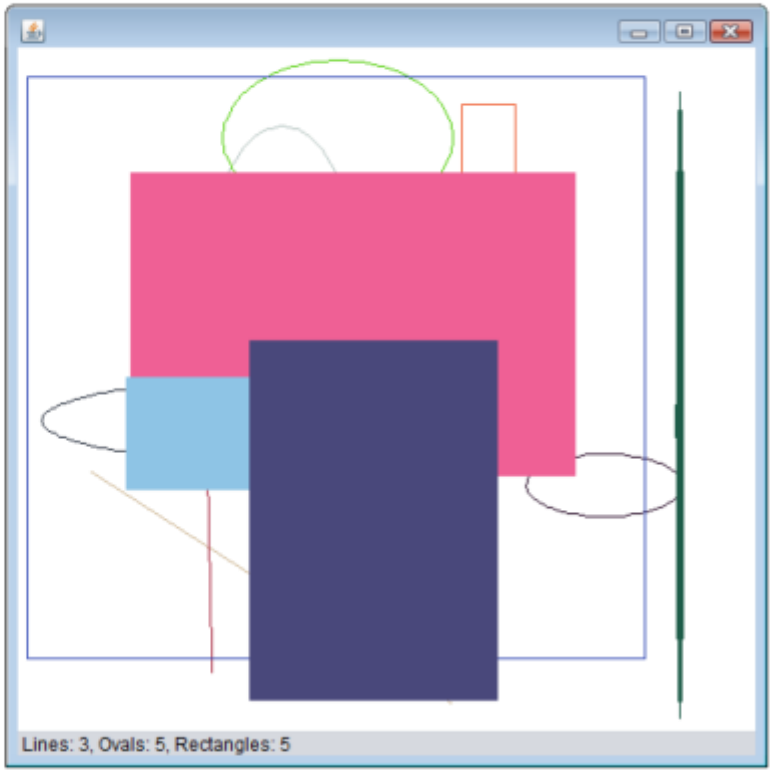

Fig. 9.14 | `JLabel` displaying shape statistics.