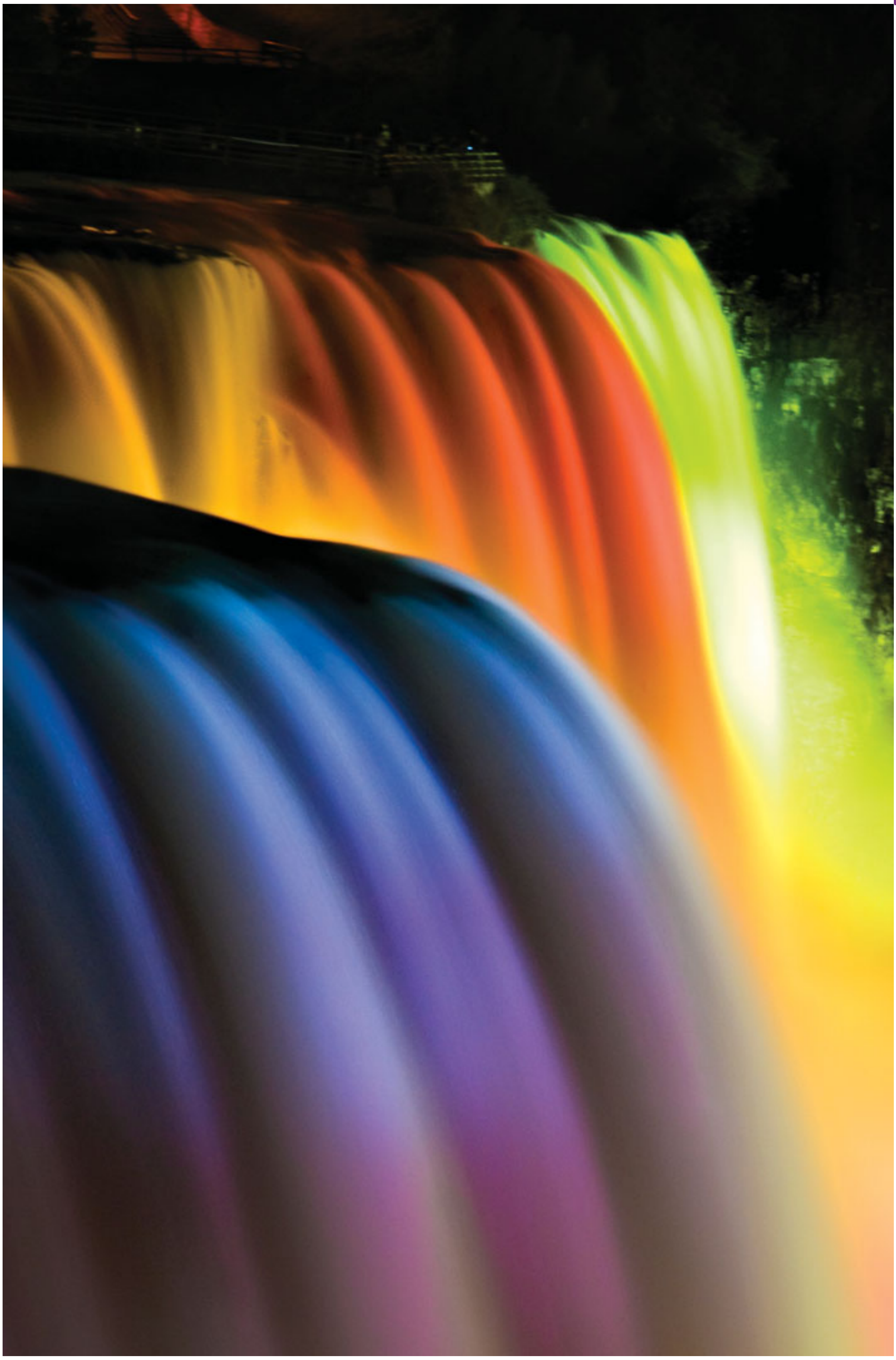
9 Object-Oriented Programming: Inheritance



Say not you know another entirely,
till you have divided an inheritance with him.
—Johann Kasper Lavater

This method is to define as the number of a class the class of all classes similar to the given class.
—Bertrand Russell

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.

Outline

9.1 Introduction

This chapter continues our discussion of object-oriented programming (OOP) by introducing one of its primary capabilities—**inheritance**, which is 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. With inheritance, you can save time during program development by basing new classes on existing proven and debugged high-quality software. This also increases the likelihood that a system will be implemented and maintained effectively.

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. The existing class is called the **superclass**, and the new class is the **subclass.** (The C++ programming language refers to the superclass as the **base class** and the subclass as the **derived class**.) Each subclass can become a superclass for future subclasses.

A subclass can add its own fields and methods. Therefore, 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 modify those behaviors so that they operate appropriately for the subclass. This is why inheritance is sometimes referred to as **specialization**.

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**, which defines the inheritance relationships between classes. In Java, the class hierarchy begins with class `Object` (in package `java.lang`), which *every* class in Java directly or indirectly **extends** (or "inherits from"). Section 9.7 lists the methods of class `Object` that are inherited by all other Java classes. Java supports only **single inheritance**, in which each class is derived from exactly *one* direct superclass. Unlike C++, Java does *not* support multiple inheritance (which occurs when a class is derived from more than one direct superclass). Chapter 10, Object-Oriented Programming: Polymorphism, explains how to use Java interfaces to realize many of the benefits of multiple inheritance while avoiding the associated problems.

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*—e.g., a car *is a* vehicle. By contrast, *has-a* represents composition (see Chapter 8). In a *has-a* relationship, *an object contains as members references to other objects*—e.g., a car *has a* steering wheel (and a car object has a reference to a steering-wheel object).

New classes can inherit from classes in **class libraries**. Organizations develop their own class libraries and can take advantage of others available worldwide. Some day, most new software likely will be constructed from **standardized reusable components**, just as automobiles and most computer hardware are constructed today. This will facilitate the development of more powerful, abundant and economical software.

9.2 Superclasses and Subclasses

Often, an object of one class *is an* object of another class as well. Figure 9.1 lists several simple examples of superclasses and subclasses—superclasses tend to be "more general" and subclasses "more specific." For example, a `CarLoan` *is a* `Loan` as are `HomeImprovementLoans` and `MortgageLoans`. Thus, in Java, class `CarLoan` can be said to inherit from class `Loan`. In this context, class `Loan` is a superclass and class `CarLoan` is a subclass. A `CarLoan` *is a* specific type of `Loan`, but it's incorrect to claim that every `Loan` *is a* `CarLoan`—the `Loan` could be any type of loan.

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.

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 often larger than the set of objects represented by any of its subclasses. For example, the superclass `Vehicle` represents all vehicles, including cars, trucks, boats, bicycles and so on. By contrast, subclass `Car` represents a smaller, more specific subset of vehicles.

University Community Member Hierarchy

Inheritance relationships form treelike hierarchical structures. A superclass exists in a hierarchical relationship with its subclasses. Let's develop a sample class hierarchy (Fig. 9.2), also called an **inheritance hierarchy**. A university community has thousands of members, including employees, students and alumni. Employees are either faculty or staff members. Faculty members are either administrators (e.g., deans and department chairpersons) or teachers. The hierarchy could contain many other classes. For example, students can be graduate or undergraduate students. Undergraduate students can be freshmen, sophomores, juniors or seniors.

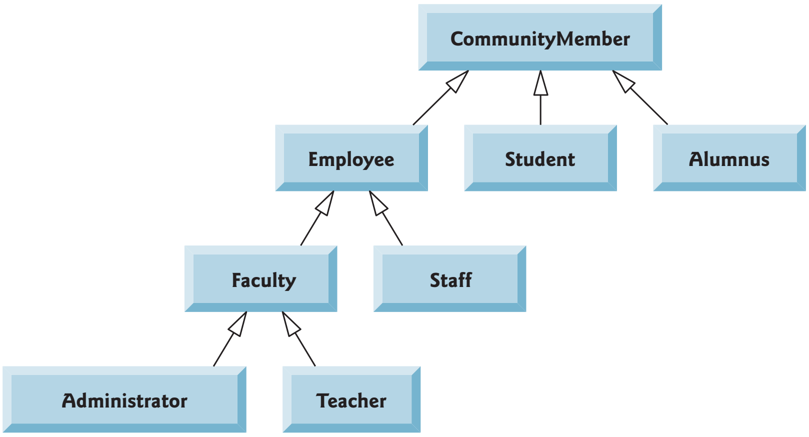


Fig. 9.2 | Inheritance hierarchy for university `CommunityMember`s.

Each arrow in the hierarchy represents an *is-a* relationship. As we follow the arrows upward in this class hierarchy, we can state, for instance, that "an `Employee` *is a* `CommunityMember`" and "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. For example, an `Administrator` *is a* `Faculty` member, *is an* `Employee`, *is a* `CommunityMember` and, of course, *is an* `Object`.

Shape Hierarchy

Now consider the `Shape` inheritance hierarchy in Fig. 9.3. This hierarchy begins with superclass `Shape`, which is extended by subclasses `TwoDimensionalShape` and `ThreeDimensionalShape`—`Shape`s are either `TwoDimensionalShape`s or `ThreeDimensionalShape`s. The third level of this hierarchy contains specific types of `TwoDimensionalShape`s and `ThreeDimensionalShape`s. As in Fig. 9.2, we can follow the arrows from the bottom of the diagram to the topmost superclass in this class hierarchy to identify several *is-a* relationships. For instance, a `Triangle` *is a* `TwoDimensionalShape` and *is a* `Shape`, while a `Sphere` *is a* `ThreeDimensionalShape` and *is a* `Shape`. This hierarchy could contain many other classes. For example, ellipses and trapezoids are `TwoDimensionalShape`s.

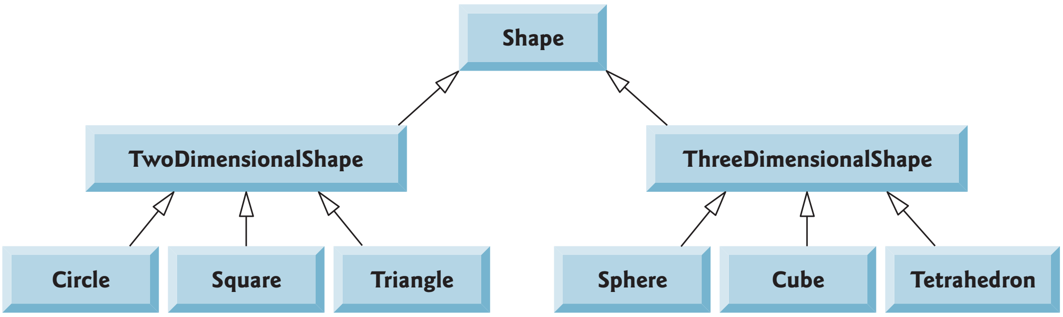


Fig. 9.3 | Inheritance hierarchy for `Shape`s.

Not every class relationship is an inheritance relationship. In Chapter 8, we discussed the *has-a* relationship, in which classes have members that are references to objects of other classes. Such relationships create classes by composition of existing classes. For 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`.

It's possible to treat superclass objects and subclass objects similarly—their commonalities are expressed in the superclass's members. Objects of all classes that extend a common superclass can be treated as objects of that superclass—such objects have an *is-a* relationship with the superclass. Later in this chapter and in Chapter 10, we consider many examples that take advantage of the *is-a* relationship.

A subclass can customize methods that it inherits from its superclass. To do this, the subclass **overrides** (redefines) the superclass method with an appropriate implementation, as we'll see often in the chapter's code examples.

9.3 protected Members

Chapter 8 discussed access modifiers `public` and `private`. 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. In this section, we introduce access modifier **`protected`**. Using `protected` access offers 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—`public` members of the superclass become `public` members of the subclass, and `protected` members of the superclass become `protected` members of the subclass. A superclass's `private` members are not accessible outside the class itself. Rather, they're *hidden* in its subclasses and 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. We discuss accessing overridden members of the superclass in Section 9.4.

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

We now use an inheritance hierarchy containing types of employees in a company's payroll application to discuss the relationship between a superclass and its subclass. In this company, commission employees (who will be represented as objects of a superclass) are paid a percentage of their sales, while base-salaried commission employees (who will be represented as objects of a subclass) receive a base salary *plus* a percentage of their sales.

We divide our discussion of the relationship between these classes into five examples. The first declares class `CommissionEmployee`, which directly inherits from class `Object` and declares as `private` instance variables a first name, last name, social security number, commission rate and gross (i.e., total) sales amount.

The second example declares class `BasePlusCommissionEmployee`, which also directly inherits from class `Object` and declares as `private` instance variables a first name, last name, social security number, commission rate, gross sales amount *and* base salary. We create this class by *writing every line of code* the class requires—we'll soon see that it's much more efficient to create it by inheriting from class `CommissionEmployee`.

The third example declares a new `BasePlusCommissionEmployee` class that *extends* class `CommissionEmployee` (i.e., a `BasePlusCommissionEmployee` *is a* `CommissionEmployee` who also has a base salary). This *software reuse lets us write less code* when developing the new subclass. In this example, class `BasePlusCommissionEmployee` attempts to access class `CommissionEmployee`'s `private` members—this results in compilation errors, because the subclass cannot access the superclass's `private` instance variables.

The fourth example shows that if `CommissionEmployee`'s instance variables are declared as `protected`, the `BasePlusCommissionEmployee` subclass can access that data directly. Both `BasePlusCommissionEmployee` classes contain identical functionality, but we show how the inherited version is easier to create and manage.

After we discuss the convenience of using `protected` instance variables, we create the fifth example, which sets the `CommissionEmployee` instance variables back to `private` to enforce good software engineering. Then we show how the `BasePlusCommissionEmployee` subclass can use `CommissionEmployee`'s `public` methods to manipulate (in a controlled manner) the `private` instance variables inherited from `CommissionEmployee`.

9.4.1 Creating and Using a `CommissionEmployee` Class

We begin by declaring class `CommissionEmployee` (Fig. 9.4). Line 4 begins the class declaration and indicates that class `CommissionEmployee` **`extends`** (i.e., inherits from) class **`Object`** (from package `java.lang`). This causes class `CommissionEmployee` to inherit the class `Object`'s methods—class `Object` does not have any fields. If you don't explicitly specify which class a new class extends, the class extends `Object` implicitly. For this reason, you typically will not include "`extends Object`" in your code—we do so in this example only for demonstration purposes.

Overview of Class `CommissionEmployee`'s Methods and Instance Variables

Class `CommissionEmployee`'s `public` services include a constructor (lines 13–22) and methods `earnings` (lines 93–96) and `toString` (lines 99–107). Lines 25–90 declare `public` *get* and *set* methods for the class's instance variables (declared in lines 6–10) `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate`. The class declares its instance variables as `private`, so objects of other classes cannot directly access these variables. Declaring instance variables as `private` and providing *get* and *set* methods to manipulate and validate them helps enforce good software engineering. Methods `setGrossSales` and `setCommissionRate`, for example, validate their arguments before assigning the values to instance variables `grossSales` and `commissionRate`. In a real-world, business-critical application, we'd also perform validation in the class's other *set* methods.

```java
// Fig. 9.4: CommissionEmployee.java
// CommissionEmployee class represents an employee paid a
// percentage of gross sales.
public class CommissionEmployee extends Object
{
   private String firstName;
   private String lastName;
   private String socialSecurityNumber;
   private double grossSales; // gross weekly sales
   private double commissionRate; // commission percentage

   // five-argument constructor
   public CommissionEmployee( String first, String last, String ssn,
      double sales, double rate )
   {
      // implicit call to Object constructor occurs here
      firstName = first;
      lastName = last;
      socialSecurityNumber = ssn;
      setGrossSales( sales ); // validate and store gross sales
      setCommissionRate( rate ); // validate and store commission rate
   } // end five-argument CommissionEmployee constructor

   // set first name
   public void setFirstName( String first )
   {
      firstName = first; // should validate
   } // end method setFirstName

   // return first name
   public String getFirstName()
   {
      return firstName;
   } // end method getFirstName

   // set last name
   public void setLastName( String last )
   {
      lastName = last; // should validate
   } // end method setLastName

   // return last name
   public String getLastName()
   {
```

Fig. 9.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 1 of 3.)

```
      return lastName;
   } // end method getLastName

   // set social security number
   public void setSocialSecurityNumber( String ssn )
   {
      socialSecurityNumber = ssn; // should validate
   } // end method setSocialSecurityNumber

   // return social security number
   public String getSocialSecurityNumber()
   {
      return socialSecurityNumber;
   } // end method getSocialSecurityNumber

   // set gross sales amount
   public void setGrossSales( double sales )
   {
      if ( sales >= 0.0 )
         grossSales = sales;
      else
         throw new IllegalArgumentException(
            "Gross sales must be >= 0.0" );
   } // end method setGrossSales

   // return gross sales amount
   public double getGrossSales()
   {
      return grossSales;
   } // end method getGrossSales

   // set commission rate
   public void setCommissionRate( double rate )
   {
      if ( rate > 0.0 && rate < 1.0 )
         commissionRate = rate;
      else
         throw new IllegalArgumentException(
            "Commission rate must be > 0.0 and < 1.0" );
   } // end method setCommissionRate

   // return commission rate
   public double getCommissionRate()
   {
      return commissionRate;
   } // end method getCommissionRate

   // calculate earnings
   public double earnings()
   {
      return commissionRate * grossSales;
   } // end method earnings
```

Fig. 9.4 | `CommissionEmployee` class represents an employee paid a percentage of gross sales. (Part 2 of 3.)

```
97
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 3 of 3.)

Class `CommissionEmployee`*'s Constructor*

Constructors are *not* inherited, so class `CommissionEmployee` does not inherit class `Object`'s constructor. However, a superclass's constructors are still available to subclasses. In fact, *the first task of any subclass constructor is to call its direct superclass's constructor*, either explicitly or implicitly (if no constructor call is specified), to ensure that the instance variables inherited from the superclass are initialized properly. In this example, class `CommissionEmployee`'s constructor calls class `Object`'s constructor implicitly. The syntax for calling a superclass constructor explicitly is discussed in Section 9.4.3. If the code does not include an explicit call to the superclass constructor, Java *implicitly* calls the superclass's default or no-argument constructor. The comment in line 16 of Fig. 9.4 indicates where the implicit call to the superclass `Object`'s default constructor is made (you do not write the code for this call). `Object`'s default (empty) constructor does nothing. Even if a class does not have constructors, the default constructor that the compiler implicitly declares for the class will call the superclass's default or no-argument constructor.

After the implicit call to `Object`'s constructor, lines 17–21 of `CommissionEmployee`'s constructor assign values to the class's instance variables. We do not validate the values of arguments `first`, `last` and `ssn` before assigning them to the corresponding instance variables. We could validate the first and last names—perhaps to ensure that they're of a reasonable length. Similarly, a social security number could be validated using regular expressions (Section 16.7) to ensure that it contains nine digits, with or without dashes (e.g., `123-45-6789` or `123456789`).

Class `CommissionEmployee`*'s* `earnings` *Method*

Method `earnings` (lines 93–96) calculates a `CommissionEmployee`'s earnings. Line 95 multiplies the `commissionRate` by the `grossSales` and returns the result.

Class `CommissionEmployee`*'s* `toString` *Method and the* `@Override` *Annotation*

Method `toString` (lines 99–107) is special—it's one of the methods that *every* class inherits directly or indirectly from class `Object` (summarized in Section 9.7). Method `toString` returns a `String` representing an object. It's called implicitly whenever an object must be converted to a `String` representation, such as when an object is output by `printf` or output by `String` method `format` via the `%s` format specifier. Class `Object`'s `toString` method returns a `String` that includes the name of the object's class. It's primarily a placeholder that can be overridden by a subclass to specify an appropriate `String` representation of the data in a subclass object. Method `toString` of class `CommissionEmployee` overrides (redefines) class `Object`'s `toString` method. When invoked, `CommissionEmployee`'s `toString` method uses `String` method `format` to return a `String` containing information about the `CommissionEmployee`. To override a superclass method, a subclass must declare a method with the same signature (method name, number of parameters, parameter types and order of parameter types) as the superclass method—`Object`'s `toString` method takes no parameters, so `CommissionEmployee` declares `toString` with no parameters.

Line 99 uses the **`@Override` annotation** to indicate that method `toString` should override a superclass method. Annotations have several purposes. For example, when you attempt to override a superclass method, common errors include naming the subclass method incorrectly, or using the wrong number or types of parameters in the parameter list. Each of these problems creates an *unintentional overload* of the superclass method. If you then attempt to call the method on a subclass object, the superclass's version is invoked and the subclass version is ignored—potentially leading to subtle logic errors. When the compiler encounters a method declared with `@Override`, it compares the method's signature with the superclass's method signatures. If there isn't an exact match, the compiler issues an error message, such as "method does not override or implement a method from a supertype." This indicates that you've accidentally overloaded a superclass method. You can then fix your method's signature so that it matches one in the superclass.

As you'll see when we discuss web applications and web services in Chapters 29–31, annotations can also add complex support code to your classes to simplify the development process and can be used by servers to configure certain aspects of web applications.

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.*

Class `CommissionEmployeeTest`

Figure 9.5 tests class `CommissionEmployee`. Lines 9–10 instantiate a `CommissionEmployee` object and invoke `CommissionEmployee`'s constructor (lines 13–22 of Fig. 9.4) to initialize it with "Sue" as the first name, "Jones" as the last name, "222-22-2222" as the social security number, 10000 as the gross sales amount and .06 as the commission rate. Lines 15–24 use CommissionEmployee's *get* methods to retrieve the object's instance-variable values for output. Lines 26–27 invoke the object's methods setGrossSales and setCommissionRate to change the values of instance variables grossSales and commissionRate. Lines 29–30 output the String representation of the updated CommissionEmployee. When an object is output using the %s format specifier, the object's toString method is invoked implicitly to obtain the object's String representation. [*Note:* In this chapter, we do not use the earnings methods of our classes—they're used extensively in Chapter 10.]

```
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() );
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
```

```
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
```

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

```
Updated employee information obtained by toString:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 500.00
commission rate: 0.10
```

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

9.4.2 Creating and Using a BasePlusCommissionEmployee Class

We now discuss the second part of our introduction to inheritance by declaring and testing (a completely new and independent) class `BasePlusCommissionEmployee` (Fig. 9.6), which contains a first name, last name, social security number, gross sales amount, commission rate *and* base salary. Class `BasePlusCommissionEmployee`'s `public` services include a `BasePlusCommissionEmployee` constructor (lines 15–25) and methods `earnings` (lines 112–115) and `toString` (lines 118–127). Lines 28–109 declare `public` *get* and *set* methods for the class's `private` instance variables (declared in lines 7–12) `firstName`, `lastName`, `socialSecurityNumber`, `grossSales`, `commissionRate` *and* `baseSalary`. These variables and methods encapsulate all the necessary features of a base-salaried commission employee. Note the *similarity* between this class and class `CommissionEmployee` (Fig. 9.4)—in this example, we'll not yet exploit that similarity.

```
 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
23        setCommissionRate( rate ); // validate and store commission rate
24        setBaseSalary( salary ); // validate and store base salary
25     } // end six-argument BasePlusCommissionEmployee constructor
```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 1 of 3.)

```java

   // set first name
   public void setFirstName( String first )
   {
      firstName = first; // should validate
   } // end method setFirstName

   // return first name
   public String getFirstName()
   {
      return firstName;
   } // end method getFirstName

   // set last name
   public void setLastName( String last )
   {
      lastName = last; // should validate
   } // end method setLastName

   // return last name
   public String getLastName()
   {
      return lastName;
   } // end method getLastName

   // set social security number
   public void setSocialSecurityNumber( String ssn )
   {
      socialSecurityNumber = ssn; // should validate
   } // end method setSocialSecurityNumber

   // return social security number
   public String getSocialSecurityNumber()
   {
      return socialSecurityNumber;
   } // end method getSocialSecurityNumber

   // set gross sales amount
   public void setGrossSales( double sales )
   {
      if ( sales >= 0.0 )
         grossSales = sales;
      else
         throw new IllegalArgumentException(
            "Gross sales must be >= 0.0" );
   } // end method setGrossSales

   // return gross sales amount
   public double getGrossSales()
   {
      return grossSales;
   } // end method getGrossSales
```

Fig. 9.6 | `BasePlusCommissionEmployee` class represents an employee who receives a base salary in addition to a commission. (Part 2 of 3.)

```java

   // set commission rate
   public void setCommissionRate( double rate )
   {
      if ( rate > 0.0 && rate < 1.0 )
         commissionRate = rate;
      else
         throw new IllegalArgumentException(
            "Commission rate must be > 0.0 and < 1.0" );
   } // end method setCommissionRate

   // return commission rate
   public double getCommissionRate()
   {
      return commissionRate;
   } // end method getCommissionRate

   // set base salary
   public void setBaseSalary( double salary )
   {
      if ( salary >= 0.0 )
         baseSalary = salary;
      else
         throw new IllegalArgumentException(
            "Base salary must be >= 0.0" );
   } // end method setBaseSalary

   // return base salary
   public double getBaseSalary()
   {
      return baseSalary;
   } // end method getBaseSalary

   // calculate earnings
   public double earnings()
   {
      return baseSalary + ( commissionRate * grossSales );
   } // end method earnings

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

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

Class `BasePlusCommissionEmployee` does not specify "`extends Object`" in line 5, so the class implicitly extends `Object`. Also, like class `CommissionEmployee`'s constructor (lines 13–22 of Fig. 9.4), class `BasePlusCommissionEmployee`'s constructor invokes class `Object`'s default constructor implicitly, as noted in the comment in line 18.

Class `BasePlusCommissionEmployee`'s `earnings` method (lines 112–115) returns the result of adding the `BasePlusCommissionEmployee`'s base salary to the product of the commission rate and the employee's gross sales.

Class `BasePlusCommissionEmployee` overrides `Object` method `toString` to return a `String` containing the `BasePlusCommissionEmployee`'s information. Once again, we use format specifier `%.2f` to format the gross sales, commission rate and base salary with two digits of precision to the right of the decimal point (line 122).

Testing Class `BasePlusCommissionEmployee`

Figure 9.7 tests class `BasePlusCommissionEmployee`. Lines 9–11 create a `BasePlusCommissionEmployee` object and pass `"Bob"`, `"Lewis"`, `"333-33-3333"`, `5000`, `.04` and `300` to the constructor as the first name, last name, social security number, gross sales, commission rate and base salary, respectively. Lines 16–27 use `BasePlusCommissionEmployee`'s *get* methods to retrieve the values of the object's instance variables for output. Line 29 invokes the object's `setBaseSalary` method to change the base salary. Method `setBaseSalary` (Fig. 9.6, lines 88–91) ensures that instance variable `baseSalary` is not assigned a negative value. Lines 31–33 of Fig. 9.7 invoke method `toString` explicitly to get the object's `String` representation.

```
 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() );
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() );
```

Fig. 9.7 | `BasePlusCommissionEmployee` test program. (Part 1 of 2.)

```

      employee.setBaseSalary( 1000 ); // set base salary

      System.out.printf( "\n%s:\n\n%s\n",
         "Updated employee information obtained by toString",
          employee.toString() );
   } // end main
} // end class BasePlusCommissionEmployeeTest
```

```
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 2 of 2.)

Notes on Class **`BasePlusCommissionEmployee`**

Much of class `BasePlusCommissionEmployee`'s code (Fig. 9.6) is similar, or identical, to that of class `CommissionEmployee` (Fig. 9.4). For example, `private` instance variables `firstName` and `lastName` and methods `setFirstName`, `getFirstName`, `setLastName` and `getLastName` are identical to those of class `CommissionEmployee`. The classes also both contain `private` instance variables `socialSecurityNumber`, `commissionRate` and `grossSales`, and corresponding *get* and *set* methods. In addition, the `BasePlusCommissionEmployee` constructor is almost identical to that of class `CommissionEmployee`, except that `BasePlusCommissionEmployee`'s constructor also sets the `baseSalary`. The other additions to class `BasePlusCommissionEmployee` are `private` instance variable `baseSalary` and methods `setBaseSalary` and `getBaseSalary`. Class `BasePlusCommissionEmployee`'s `toString` method is nearly identical to that of class `CommissionEmployee` except that it also outputs instance variable `baseSalary` with two digits of precision to the right of the decimal point.

We literally *copied* code from class `CommissionEmployee` and *pasted* it into class `BasePlusCommissionEmployee`, then modified class `BasePlusCommissionEmployee` to include a base salary and methods that manipulate the base salary. This *"copy-and-paste" approach* is often error prone and time consuming. Worse yet, it spreads copies of the same code throughout a system, creating a code-maintenance nightmare. Is there a way to "absorb" the instance variables and methods of one class in a way that makes them part of other classes *without duplicating code*? Next we answer this question, using a more elegant approach to building classes that emphasizes the benefits of inheritance.

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

Now we redeclare class BasePlusCommissionEmployee (Fig. 9.8) to *extend* class CommissionEmployee (Fig. 9.4). A BasePlusCommissionEmployee object *is a* CommissionEmployee, because inheritance passes on class CommissionEmployee's capabilities. Class BasePlusCommissionEmployee also has instance variable baseSalary (Fig. 9.8, line 6). Keyword extends (line 4) indicates inheritance. BasePlusCommissionEmployee *inherits* CommissionEmployee's instance variables and methods, but only the superclass's public and protected members are directly accessible in the subclass. The CommissionEmployee constructor is *not* inherited. So, the public BasePlusCommissionEmployee services include its constructor (lines 9–16), public methods inherited from CommissionEmployee, and methods setBaseSalary (lines 19–26), getBaseSalary (lines 29–32), earnings (lines 35–40) and toString (lines 43–53). Methods earnings and toString *override* the corresponding methods in class CommissionEmployee because their superclass versions do not properly calculate a BasePlusCommissionEmployee's earnings or return an appropriate String representation.

```java
 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        if ( salary >= 0.0 )
22           baseSalary = salary;
23        else
24           throw new IllegalArgumentException(
25              "Base salary must be >= 0.0" );
26     } // end method setBaseSalary
```

Fig. 9.8 | private superclass members cannot be accessed in a subclass. (Part 1 of 3.)

```java

   // return base salary
   public double getBaseSalary()
   {
      return baseSalary;
   } // end method getBaseSalary

   // calculate earnings
   @Override // indicates that this method overrides a superclass method
   public double earnings()
   {
      // not allowed: commissionRate and grossSales private in superclass
      return baseSalary + ( commissionRate * grossSales );
   } // end method earnings

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

```
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 2 of 3.)

```
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 3 of 3.)

A Subclass's Constructor Must Call Its Superclass's Constructor

Each subclass constructor must implicitly or explicitly call its superclass constructor to initialize the instance variables inherited from the superclass. Line 13 in `BasePlusCommissionEmployee`'s six-argument constructor (lines 9–16) explicitly calls class `CommissionEmployee`'s five-argument constructor (declared at lines 13–22 of Fig. 9.4) to initialize the superclass portion of a `BasePlusCommissionEmployee` object (i.e., variables `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate`). We do this by using the **superclass constructor call syntax**—keyword `super`, followed by a set of parentheses containing the superclass constructor arguments. The arguments `first`, `last`, `ssn`, `sales` and `rate` are used to initialize superclass members `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate`, respectively. If `BasePlusCommissionEmployee`'s 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. The explicit superclass constructor call in line 13 of Fig. 9.8 must be the *first* statement in the subclass constructor's body. When a superclass contains a no-argument constructor, you can use `super()` to call that constructor explicitly, but this is rarely done.

`BasePlusCommissionEmployee` Method `Earnings`

The compiler generates errors for line 39 because superclass `CommissionEmployee`'s instance variables `commissionRate` and `grossSales` are `private`—subclass `BasePlusCommissionEmployee`'s methods are not allowed to access superclass `CommissionEmployee`'s `private` instance variables. We used red text in Fig. 9.8 to indicate erroneous code. The compiler issues additional errors at lines 49–51 of `BasePlusCommissionEmployee`'s `toString` method for the same reason. The errors in `BasePlusCommissionEmployee` could have been prevented by using the *get* methods inherited from class `CommissionEmployee`. For example, line 39 could have used `getCommissionRate` and `getGrossSales` to access `CommissionEmployee`'s `private` instance variables `commissionRate` and `grossSales`, respectively. Lines 49–51 also 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 class `BasePlusCommissionEmployee` to directly access superclass instance variables `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate`, we can declare those members as `protected` in the superclass. As we discussed in Section 9.3, a superclass's `protected` members are accessible by all subclasses of that superclass. In the new `CommissionEmployee` class, we modified only lines 6–10 of Fig. 9.4 to declare the instance variables with the `protected` access modifier as follows:

```
protected String firstName;
protected String lastName;
protected String socialSecurityNumber;
protected double grossSales; // gross weekly sales
protected double commissionRate; // commission percentage
```

The rest of the class declaration (which is not shown here) is identical to that of Fig. 9.4.

We could have declared `CommissionEmployee`'s instance variables `public` to enable subclass `BasePlusCommissionEmployee` to access them. However, declaring `public` instance variables is poor software engineering because it allows unrestricted access to the these variables, greatly increasing the chance of errors. 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—recall that `protected` class members are also visible to other classes in the same package.

Class `BasePlusCommissionEmployee`

Class `BasePlusCommissionEmployee` (Fig. 9.9) extends the new version of class `CommissionEmployee` with `protected` instance variables. `BasePlusCommissionEmployee` objects inherit `CommissionEmployee`'s `protected` instance variables `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate`—all these variables are now `protected` members of `BasePlusCommissionEmployee`. As a result, the compiler does not generate errors when compiling line 37 of method `earnings` and lines 46–48 of method `toString`. If another class extends this version of class `BasePlusCommissionEmployee`, the new subclass also can access the `protected` members.

When you create a `BasePlusCommissionEmployee` object, it contains all instance variables declared in the class hierarchy to that point—i.e., those from classes `Object`, `CommissionEmployee` and `BasePlusCommissionEmployee`. Class `BasePlusCommissionEmployee` does not inherit class `CommissionEmployee`'s constructor. However, class `BasePlusCommissionEmployee`'s six-argument constructor (lines 10–15) calls class `CommissionEmployee`'s five-argument constructor *explicitly* to initialize the instance variables that `BasePlusCommissionEmployee` inherited from class `CommissionEmployee`. Similarly, class `CommissionEmployee`'s constructor *implicitly* calls class `Object`'s constructor. `BasePlusCommissionEmployee`'s constructor must do this *explicitly* because `CommissionEmployee` does *not* provide a no-argument constructor that could be invoked implicitly.

```
 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     {
```

Fig. 9.9 | `BasePlusCommissionEmployee` inherits `protected` instance variables from `CommissionEmployee`. (Part 1 of 2.)

```
      super( first, last, ssn, sales, rate );
      setBaseSalary( salary ); // validate and store base salary
   } // end six-argument BasePlusCommissionEmployee constructor

   // set base salary
   public void setBaseSalary( double salary )
   {
      if ( salary >= 0.0 )
         baseSalary = salary;
      else
         throw new IllegalArgumentException(
            "Base salary must be >= 0.0" );
   } // end method setBaseSalary

   // return base salary
   public double getBaseSalary()
   {
      return baseSalary;
   } // end method getBaseSalary

   // calculate earnings
   @Override // indicates that this method overrides a superclass method
   public double earnings()
   {
      return baseSalary + ( commissionRate * grossSales );
   } // end method earnings

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

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

Testing Class `BasePlusCommissionEmployee`

The `BasePlusCommissionEmployeeTest` class for this example is identical to that of Fig. 9.7 and produces the same output, so we do not show it here. Although the version of class `BasePlusCommissionEmployee` in Fig. 9.6 does not use inheritance and the version in Fig. 9.9 does, *both classes provide the same functionality*. The source code in Fig. 9.9 (47 lines) is considerably shorter than that in Fig. 9.6 (116 lines), because most of `BasePlusCommissionEmployee`'s functionality is now inherited from `CommissionEmployee`—there's now only one copy of the `CommissionEmployee` functionality. This makes the code easier to maintain, modify and debug, because the code related to a commission employee exists only in class `CommissionEmployee`.

Notes on Using **protected** *Instance Variables*

In this example, we declared superclass instance variables as `protected` so that subclasses could access them. 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, however, it's better to use `private` instance variables to encourage proper software engineering, and leave code optimization issues to the compiler. Your code will be easier to maintain, modify and debug.

Using `protected` instance variables creates several potential problems. First, the subclass object can set an inherited variable's value directly without using a *set* method. Therefore, a subclass object can assign an invalid value to the variable, possibly leaving the object in an inconsistent state. For example, if we were to declare `CommissionEmployee`'s instance variable `grossSales` as `protected`, a subclass object (e.g., `BasePlusCommissionEmployee`) could then assign a negative value to `grossSales`. Another problem with using `protected` instance variables is that subclass methods are more likely to be written so that they depend on the superclass's data implementation. In practice, subclasses should depend only on the superclass services (i.e., non-`private` methods) and not on the superclass data implementation. With `protected` instance variables in the superclass, we may need to modify all the subclasses of the superclass if the superclass implementation changes. For example, if for some reason we were to change the names of instance variables `firstName` and `lastName` to `first` and `last`, then we would have to do so for all occurrences in which a subclass directly references superclass instance variables `firstName` and `lastName`. In such a case, the 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. Of course, if the superclass services change, we must reimplement our subclasses. A third problem is that 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`–`BasePlusCommissionEmployee` Inheritance Hierarchy Using `private` Instance Variables

Let's reexamine our hierarchy once more, this time using good software engineering practices. Class `CommissionEmployee` (Fig. 9.10) declares instance variables `firstName`, `lastName`, socialSecurityNumber, grossSales and commissionRate as *private* (lines 6–10) and provides public methods setFirstName, getFirstName, setLastName, getLastName, setSocialSecurityNumber, getSocialSecurityNumber, setGrossSales, getGrossSales, setCommissionRate, getCommissionRate, earnings and toString for manipulating these values. Methods earnings (lines 93–96) and toString (lines 99–107) use the class's *get* methods to obtain the values of its instance variables. If we decide to change the instance-variable names, the earnings and toString declarations will not require modification—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.

```java
 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
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     {
```

Fig. 9.10 | CommissionEmployee class uses methods to manipulate its private instance variables. (Part 1 of 3.)

```
      la5stName = last; // should validate
   } // end method setLastName

   // return last name
   public String getLastName()
   {
      return lastName;
   } // end method getLastName

   // set social security number
   public void setSocialSecurityNumber( String ssn )
   {
      socialSecurityNumber = ssn; // should validate
   } // end method setSocialSecurityNumber

   // return social security number
   public String getSocialSecurityNumber()
   {
      return socialSecurityNumber;
   } // end method getSocialSecurityNumber

   // set gross sales amount
   public void setGrossSales( double sales )
   {
      if ( sales >= 0.0 )
         grossSales = sales;
      else
         throw new IllegalArgumentException(
            "Gross sales must be >= 0.0" );
   } // end method setGrossSales

   // return gross sales amount
   public double getGrossSales()
   {
      return grossSales;
   } // end method getGrossSales

   // set commission rate
   public void setCommissionRate( double rate )
   {
      if ( rate > 0.0 && rate < 1.0 )
         commissionRate = rate;
      else
         throw new IllegalArgumentException(
            "Commission rate must be > 0.0 and < 1.0" );
   } // end method setCommissionRate

   // return commission rate
   public double getCommissionRate()
   {
      return commissionRate;
   } // end method getCommissionRate
```

Fig. 9.10 | `CommissionEmployee` class uses methods to manipulate its `private` instance variables. (Part 2 of 3.)

```

   // calculate earnings
   public double earnings()
   {
      return getCommissionRate() * getGrossSales();
   } // end method earnings

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

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

Subclass `BasePlusCommissionEmployee` (Fig. 9.11) inherits `CommissionEmployee`'s non-`private` methods and can access the `private` superclass members via those methods. Class `BasePlusCommissionEmployee` has several changes that distinguish it from Fig. 9.9. Methods `earnings` (lines 35–39) and `toString` (lines 42–47) each invoke method `getBaseSalary` to obtain the base salary value, rather than accessing `baseSalary` directly. If we decide to rename instance variable `baseSalary`, only the bodies of method `setBaseSalary` and `getBaseSalary` will need to change.

```
 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 2.)

```
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
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 2 of 2.)

Class `BasePlusCommissionEmployee`*'s* `earnings` *Method*

Method `earnings` (lines 35–39) overrides class `CommissionEmployee`'s `earnings` method (Fig. 9.10, lines 93–96) to calculate a base-salaried commission employee's earnings. The new version obtains the portion of the earnings based on commission alone by calling `CommissionEmployee`'s `earnings` method with `super.earnings()` (line 34), then adds the base salary to this value to calculate the total earnings. Note the syntax used to invoke an overridden superclass method from a subclass—place the keyword `super` and a dot (`.`) separator before the superclass method name. This method invocation is a 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. By having `BasePlusCommissionEmployee`'s `earnings` method invoke `CommissionEmployee`'s `earnings` method to calculate part of a `BasePlusCommissionEmployee` object's earnings, we *avoid duplicating the code* and *reduce code-maintenance problems*. If we did not use "`super.`" then `BasePlusCommissionEmployee`'s `earnings` method would *call itself* rather than the superclass version. This would result in a phenomenon we study in Chapter 18 called *infinite recursion*, which would eventually cause the method-call stack to overflow—a fatal runtime error.

Class `BasePlusCommissionEmployee`*'s* `toString` *Method*

Similarly, `BasePlusCommissionEmployee`'s `toString` method (Fig. 9.11, lines 38–43) overrides class `CommissionEmployee`'s `toString` method (Fig. 9.10, lines 91–99) to return a `String` representation that's appropriate for a base-salaried commission employee. The new version creates part of a `BasePlusCommissionEmployee` object's `String` representation (i.e., the `String` `"commission employee"` and the values of class `CommissionEmployee`'s `private` instance variables) by calling `CommissionEmployee`'s `toString` method with the expression `super.toString()` (Fig. 9.11, line 42). `BasePlusCommissionEmployee`'s `toString` method then outputs the remainder of a `BasePlusCommissionEmployee` object's `String` representation (i.e., the value of class `BasePlusCommissionEmployee`'s base salary).

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.*

Testing Class `BasePlusCommissionEmployee`

Class `BasePlusCommissionEmployeeTest` performs the same manipulations on a `BasePlusCommissionEmployee` object as in Fig. 9.7 and produces the same output, so we do not show it here. Although each `BasePlusCommissionEmployee` class you've seen behaves identically, the version in Fig. 9.11 is the best engineered. By using inheritance and by calling methods that hide the data and ensure consistency, we've efficiently and effectively constructed a well-engineered class.

Summary of the Inheritance Examples in Sections 9.4.1–9.4.5

You've now seen a set of examples that were designed to teach good software engineering with inheritance. You used the keyword `extends` to create a subclass using inheritance, used `protected` superclass members to enable a subclass to access inherited superclass instance variables, and overrode superclass methods to provide versions that are more appropriate for subclass objects. In addition, you applied software engineering techniques from Chapter 8 and this chapter to create classes that are easy to maintain, modify and debug.

9.5 Constructors in Subclasses

As we explained in the preceding section, instantiating a subclass object begins a chain of constructor calls in which the subclass constructor, before performing its own tasks, invokes its direct superclass's constructor either explicitly via the `super` reference or implicitly calling the superclass's default constructor or no-argument constructor. Similarly, if the superclass is derived from another class—as is, of course, every class except `Object`—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* the constructor for class `Object`. The original subclass constructor's body finishes executing *last*. Each superclass's constructor manipulates the superclass instance variables that the subclass object inherits. For example, consider again the `CommissionEmployee`–`BasePlusCommissionEmployee` hierarchy from Fig. 9.10 and Fig. 9.11. When a program creates a `BasePlusCommissionEmployee` object, its constructor is called. That constructor calls `CommissionEmployee`'s constructor, which in turn calls `Object`'s constructor. Class `Object`'s constructor has an empty body, so it immediately returns control to `CommissionEmployee`'s constructor, which then initializes the `CommissionEmployee` `private` instance variables that are part of the `BasePlusCommissionEmployee` object. When `CommissionEmployee`'s constructor completes execution, it returns control to `BasePlusCommissionEmployee`'s constructor, which initializes the `BasePlusCommissionEmployee` object's `baseSalary`.

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 so it can compile and execute any program that uses or extends the superclass. This powerful capability is attractive to independent software vendors (ISVs), who can develop proprietary classes for sale or license and make them available to users in bytecode format. Users then can derive new classes from these library classes rapidly and without accessing the ISVs' proprietary source code.

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.

It's sometimes difficult to appreciate the scope of the problems faced by designers who work on large-scale software projects. People experienced with such projects say that effective software reuse improves the software-development process. Object-oriented programming facilitates software reuse, often significantly shortening development time.

The availability of substantial and useful class libraries delivers the maximum benefits of software reuse through inheritance. The standard Java class libraries that are shipped with Java tend to be rather general purpose, encouraging broad software reuse. Many other class libraries exist.

Reading subclass declarations can be confusing, because inherited members are not declared explicitly in the subclasses but are nevertheless present in them. A similar problem exists in documenting subclass members.

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 Class `Object`

As we discussed earlier in this chapter, all classes in Java inherit directly or indirectly from the `Object` class (package `java.lang`), so its 11 methods (some are overloaded) are inherited by all other classes. Figure 9.12 summarizes `Object`'s methods. We discuss several `Object` methods throughout this book (as indicated in Fig. 9.12).

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.
`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 1 of 2.)

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 2 of 2.)

Recall from Chapter 7 that arrays are objects. As a result, like all other objects, arrays inherit the members of class `Object`. Every array has an overridden `clone` method that copies the array. However, if the array stores references to objects, the objects are not copied—a *shallow copy is* performed.

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

Programs often use labels when they need to display information or instructions to the user in a graphical user interface. **Labels** are a convenient way of identifying GUI components on the screen and keeping the user informed about the current state of the program. In Java, an object of class **`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.

Lines 3–6 import the classes we need to display `JLabel`s. `BorderLayout` from package `java.awt` contains constants that specify where we can place GUI components in the `JFrame`. Class **`ImageIcon`** represents an image that can be displayed on a `JLabel`, and class `JFrame` represents the window that will contain all the labels.

```
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
```

Fig. 9.13 | `JLabel` with text and with images. (Part 1 of 2.)

```
 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
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
```

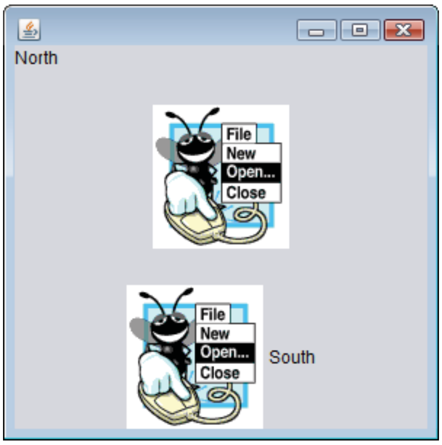


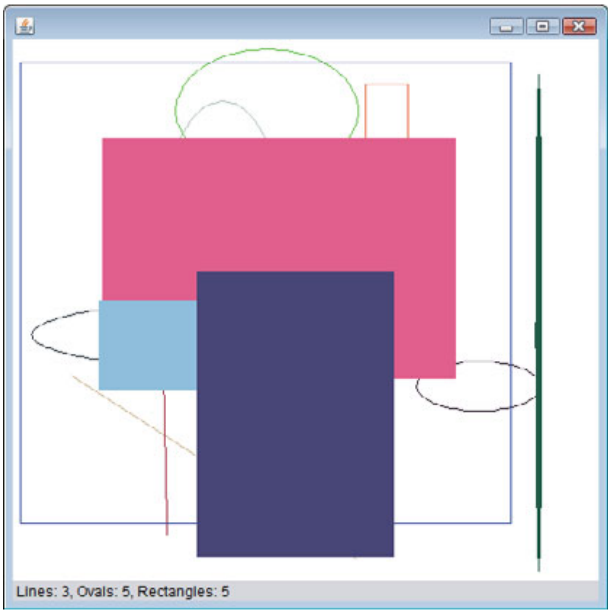
Fig. 9.13 | `JLabel` with text and with images. (Part 2 of 2.)

Line 13 creates a `JLabel` that displays its constructor argument—the string `"North"`. Line 16 declares local variable `labelIcon` and assigns it a new `ImageIcon`. The constructor for ImageIcon receives a String that specifies the path to the image. Since we specify only a file name, Java assumes that it's in the same directory as class LabelDemo. ImageIcon can load images in GIF, JPEG and PNG image formats. Line 19 declares and initializes local variable centerLabel with a JLabel that displays the labelIcon. Line 22 declares and initializes local variable southLabel with a JLabel similar to the one in line 19. However, line 25 calls method **setText** to change the text the label displays. Method setText can be called on any JLabel to change its text. This JLabel displays both the icon and the text.

Line 28 creates the JFrame that displays the JLabels, and line 30 indicates that the program should terminate when the JFrame is closed. We attach the labels to the JFrame in lines 34–36 by calling an overloaded version of method add that takes two parameters. The first parameter is the component we want to attach, and the second is the region in which it should be placed. Each JFrame has an associated **layout** that helps the JFrame position the GUI components that are attached to it. The default layout for a JFrame is known as a **BorderLayout** and has five regions—NORTH (top), SOUTH (bottom), EAST (right side), WEST (left side) and CENTER. Each of these is declared as a constant in class BorderLayout. When calling method add with one argument, the JFrame places the component in the CENTER automatically. If a position already contains a component, then the new component takes its place. Lines 38 and 39 set the size of the JFrame and make it visible on screen.

GUI and Graphics Case Study Exercise

9.1 Modify GUI and Graphics Case Study Exercise 8.1 to include a JLabel as a status bar that displays counts representing the number of each shape displayed. Class DrawPanel should declare a method that returns a String containing the status text. In main, first create the DrawPanel, then create the JLabel with the status text as an argument to the JLabel's constructor. Attach the JLabel to the SOUTH region of the JFrame, as shown in Fig. 9.14.



Lines: 3, Ovals: 5, Rectangles: 5

Fig. 9.14 | JLabel displaying shape statistics.

9.9 Wrap-Up

This chapter introduced inheritance—the ability to create classes by absorbing an existing class's members and embellishing them with new capabilities. You learned the notions of superclasses and subclasses and used keyword `extends` to create a subclass that inherits members from a superclass. We showed how to use the `@Override` annotation to prevent unintended overloading by indicating that a method overrides a superclass method. We introduced the access modifier `protected`; subclass methods can directly access `protected` superclass members. You learned how to use `super` to access overridden superclass members. You also saw how constructors are used in inheritance hierarchies. Finally, you learned about the methods of class `Object`, the direct or indirect superclass of all Java classes.

In Chapter 10, Object-Oriented Programming: Polymorphism, we build on our discussion of inheritance by introducing polymorphism—an object-oriented concept that enables us to write programs that conveniently handle, in a more general manner, objects of a wide variety of classes related by inheritance. After studying Chapter 10, you'll be familiar with classes, objects, encapsulation, inheritance and polymorphism—the key technologies of object-oriented programming.

Summary

Section 9.1 Introduction

- Inheritance (p. 360) reduces program-development time.
- The direct superclass (p. 360) of a subclass (specified by the keyword `extends` in the first line of a class declaration) is the superclass from which the subclass inherits. An indirect superclass (p. 360) of a subclass is two or more levels up the class hierarchy from that subclass.
- In single inheritance (p. 360), a class is derived from one direct superclass. In multiple inheritance, a class is derived from more than one direct superclass. Java does not support multiple inheritance.
- A subclass is more specific than its superclass and represents a smaller group of objects (p. 360).
- Every object of a subclass is also an object of that class's superclass. However, a superclass object is not an object of its class's subclasses.
- An *is-a* relationship (p. 361) represents inheritance. In an *is-a* relationship, an object of a subclass also can be treated as an object of its superclass.
- A *has-a* relationship (p. 361) represents composition. In a *has-a* relationship, a class object contains references to objects of other classes.

Section 9.2 Superclasses and Subclasses

- Single-inheritance relationships form treelike hierarchical structures—a superclass exists in a hierarchical relationship with its subclasses.

Section 9.3 `protected` Members

- A superclass's `public` members are accessible wherever the program has a reference to an object of that superclass or one of its subclasses.
- A superclass's `private` members can be accessed directly only within the superclass's declaration.
- A superclass's `protected` members (p. 363) have an intermediate level of protection between `public` and `private` access. They can be accessed by members of the superclass, by members of its subclasses and by members of other classes in the same package.

- A superclass's `private` members are hidden in its subclasses and can be accessed only through the `public` or `protected` methods inherited from the superclass.
- An overridden superclass method can be accessed from a subclass if the superclass method name is preceded by `super` (p. 363) and a dot (`.`) separator.

Section 9.4 Relationship between Superclasses and Subclasses

- A subclass cannot access the `private` members of its superclass, but it can access the non-`private` members.
- A subclass can invoke a constructor of its superclass by using the keyword `super`, followed by a set of parentheses containing the superclass constructor arguments. This must appear as the first statement in the subclass constructor's body.
- A superclass method can be overridden in a subclass to declare an appropriate implementation for the subclass.
- The annotation `@Override` (p. 368) indicates that a method should override a superclass method. When the compiler encounters a method declared with `@Override`, it compares the method's signature with the superclass's method signatures. If there isn't an exact match, the compiler issues an error message, such as "method does not override or implement a method from a supertype."
- Method `toString` takes no arguments and returns a `String`. The `Object` class's `toString` method is normally overridden by a subclass.
- When an object is output using the `%s` format specifier, the object's `toString` method is called implicitly to obtain its `String` representation.

Section 9.5 Constructors in Subclasses

- The first task of a subclass constructor is to call its direct superclass's constructor (p. 377) to ensure that the instance variables inherited from the superclass are initialized.

Section 9.6 Software Engineering with Inheritance

- Declaring instance variables `private`, while providing non-`private` methods to manipulate and perform validation, helps enforce good software engineering.

Section 9.7 `Object` Class

- See the table of class `Object`'s methods in Fig. 9.12.

Self-Review Exercises

9.1 Fill in the blanks in each of the following statements:

a) ________ is a form of software reusability in which new classes acquire the members of existing classes and embellish those classes with new capabilities.

b) A superclass's ________ members can be accessed in the superclass declaration *and in* subclass declarations.

c) In a(n) ________ relationship, an object of a subclass can also be treated as an object of its superclass.

d) In a(n) ________ relationship, a class object has references to objects of other classes as members.

e) In single inheritance, a class exists in a(n) ________ relationship with its subclasses.

f) A superclass's ________ members are accessible anywhere that the program has a reference to an object of that superclass or to an object of one of its subclasses.

g) When an object of a subclass is instantiated, a superclass ________ is called implicitly or explicitly.

h) Subclass constructors can call superclass constructors via the ________ keyword.

9.2 State whether each of the following is *true* or *false*. If a statement is *false*, explain why.

a) Superclass constructors are not inherited by subclasses.
b) A *has-a* relationship is implemented via inheritance.
c) A `Car` class has an *is-a* relationship with the `SteeringWheel` and `Brakes` classes.
d) When a subclass redefines a superclass method by using the same signature, the subclass is said to overload that superclass method.

Answers to Self-Review Exercises

9.1 a) Inheritance. b) `public` and `protected`. c) *is-a* or inheritance. d) *has-a* or composition. e) hierarchical. f) `public`. g) constructor. h) `super`.

9.2 a) True. b) False. A *has-a* relationship is implemented via composition. An *is-a* relationship is implemented via inheritance. c) False. This is an example of a *has-a* relationship. Class `Car` has an *is-a* relationship with class `Vehicle`. d) False. This is known as overriding, not overloading—an overloaded method has the same name, but a different signature.

Exercises

9.3 Many programs written with inheritance could be written with composition instead, and vice versa. Rewrite class `BasePlusCommissionEmployee` (Fig. 9.11) of the `CommissionEmployee`–`BasePlusCommissionEmployee` hierarchy to use composition rather than inheritance.

9.4 Discuss the ways in which inheritance promotes software reuse, saves time during program development and helps prevent errors.

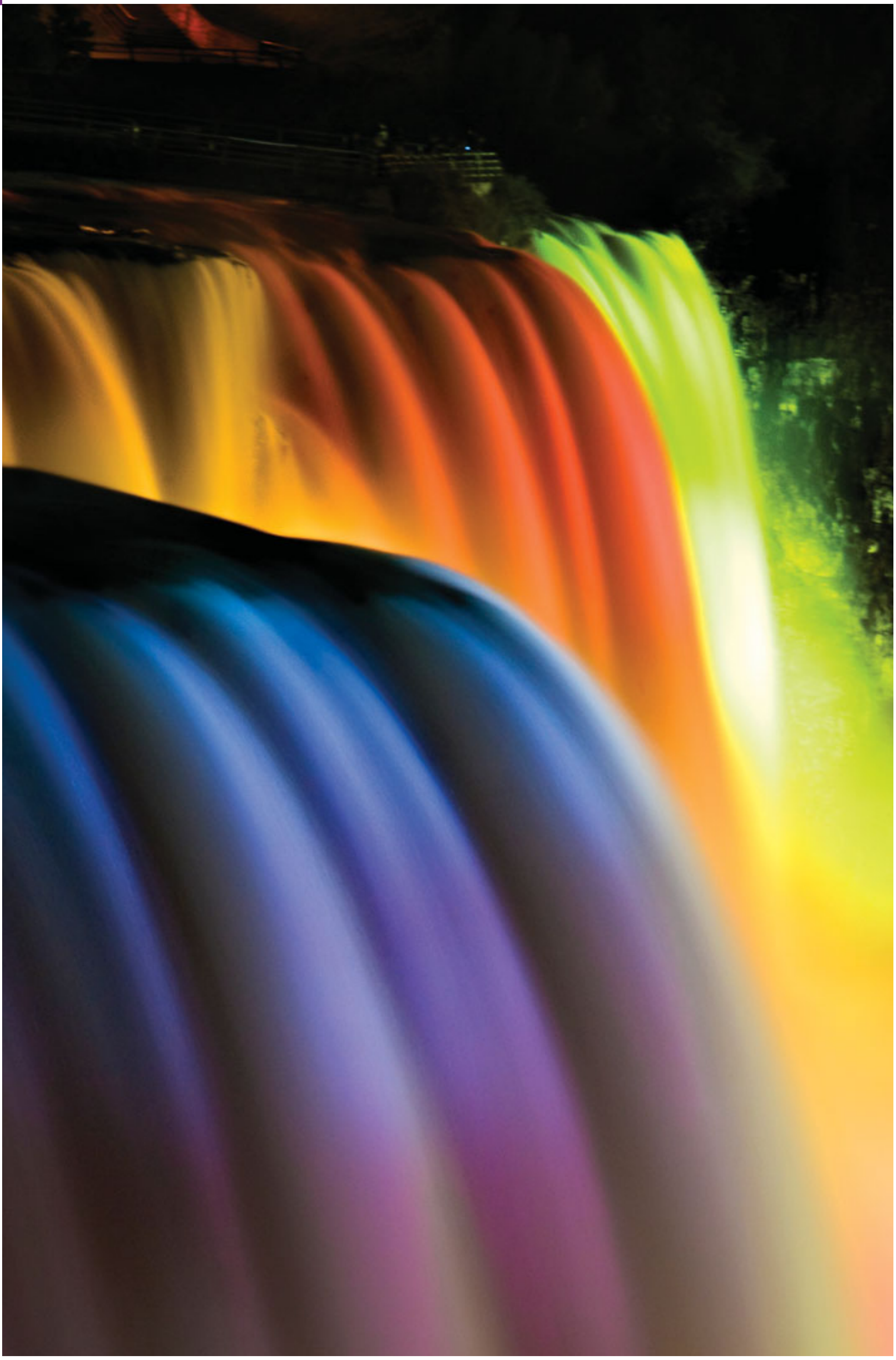
9.5 Draw an inheritance hierarchy for students at a university similar to the hierarchy shown in Fig. 9.2. Use `Student` as the superclass of the hierarchy, then extend `Student` with classes `UndergraduateStudent` and `GraduateStudent`. Continue to extend the hierarchy as deep (i.e., as many levels) as possible. For example, `Freshman`, `Sophomore`, `Junior` and `Senior` might extend `UndergraduateStudent`, and `DoctoralStudent` and `MastersStudent` might be subclasses of `GraduateStudent`. After drawing the hierarchy, discuss the relationships that exist between the classes. [*Note:* You do not need to write any code for this exercise.]

9.6 The world of shapes is much richer than the shapes included in the inheritance hierarchy of Fig. 9.3. Write down all the shapes you can think of—both two-dimensional and three-dimensional—and form them into a more complete `Shape` hierarchy with as many levels as possible. Your hierarchy should have class `Shape` at the top. Classes `TwoDimensionalShape` and `ThreeDimensionalShape` should extend `Shape`. Add additional subclasses, such as `Quadrilateral` and `Sphere`, at their correct locations in the hierarchy as necessary.

9.7 Some programmers prefer not to use `protected` access, because they believe it breaks the encapsulation of the superclass. Discuss the relative merits of using `protected` access vs. using `private` access in superclasses.

9.8 Write an inheritance hierarchy for classes `Quadrilateral`, `Trapezoid`, `Parallelogram`, `Rectangle` and `Square`. Use `Quadrilateral` as the superclass of the hierarchy. Create and use a `Point` class to represent the points in each shape. Make the hierarchy as deep (i.e., as many levels) as possible. Specify the instance variables and methods for each class. The `private` instance variables of `Quadrilateral` should be the *x-y* coordinate pairs for the four endpoints of the `Quadrilateral`. Write a program that instantiates objects of your classes and outputs each object's area (except `Quadrilateral`).

10 Object-Oriented Programming: Polymorphism



One Ring to rule them all,
One Ring to find them,
One Ring to bring them all
and in the darkness bind them.
—John Ronald Reuel Tolkien

General propositions do not decide concrete cases.
—Oliver Wendell Holmes

A philosopher of imposing stature doesn't think in a vacuum. Even his most abstract ideas are, to some extent, conditioned by what is or is not known in the time when he lives.
—Alfred North Whitehead

Why art thou cast down, O my soul?
—Psalms 42:5

Objectives

In this chapter you'll learn:

- The concept of polymorphism.
- To use overridden methods to effect polymorphism.
- To distinguish between abstract and concrete classes.
- To declare abstract methods to create abstract classes.
- How polymorphism makes systems extensible and maintainable.
- To determine an object's type at execution time.
- To declare and implement interfaces.

Outline

10.1 Introduction

We continue our study of object-oriented programming by explaining and demonstrating **polymorphism** with inheritance hierarchies. Polymorphism enables you to “program in the general” rather than “program in the specific.” In particular, polymorphism enables you to write programs that process objects that share the same superclass (either directly or indirectly) as if they’re all objects of the superclass; this can simplify programming.

Consider the following example of polymorphism. Suppose we create a program that simulates the movement of several types of animals for a biological study. Classes `Fish`, `Frog` and `Bird` represent the types of animals under investigation. Imagine that each class extends superclass `Animal`, which contains a method `move` and maintains an animal’s current location as *x-y* coordinates. Each subclass implements method `move`. Our program maintains an `Animal` array containing references to objects of the various `Animal` subclasses. To simulate the animals’ movements, the program sends each object the *same* message once per second—namely, `move`. Each specific type of `Animal` responds to a `move` message in its own way—a `Fish` might swim three feet, a `Frog` might jump five feet and a `Bird` might fly ten feet. Each object knows how to modify its *x-y* coordinates appropriately for its *specific* type of movement. Relying on each object to know how to “do the right thing” (i.e., do what is appropriate for that type of object) in response to the same method call is the key concept of polymorphism. The same message (in this case, `move`) sent to a variety of objects has “many forms” of results—hence the term polymorphism.

Implementing for Extensibility

With polymorphism, we can design and implement systems that are easily extensible—new classes can be added with little or no modification to the general portions of the program, as long as the new classes are part of the inheritance hierarchy that the program processes generically. The only parts of a program that must be altered are those that require direct knowledge of the new classes that we add to the hierarchy. For example, if we extend class `Animal` to create class `Tortoise` (which might respond to a `move` message by crawling one inch), we need to write only the `Tortoise` class and the part of the simulation that instantiates a `Tortoise` object. The portions of the simulation that tell each `Animal` to move generically can remain the same.

Chapter Overview

First, we discuss common examples of polymorphism. We then provide a simple example demonstrating polymorphic behavior. We use superclass references to manipulate *both* superclass objects and subclass objects polymorphically.

We then present a case study that revisits the employee hierarchy of Section 9.4.5. We develop a simple payroll application that polymorphically calculates the weekly pay of several different types of employees using each employee's `earnings` method. Though the earnings of each type of employee are calculated in a specific way, polymorphism allows us to process the employees "in the general." In the case study, we enlarge the hierarchy to include two new classes—`SalariedEmployee` (for people paid a fixed weekly salary) and `HourlyEmployee` (for people paid an hourly salary and "time-and-a-half" for overtime). We declare a common set of functionality for all the classes in the updated hierarchy in an "abstract" class, `Employee`, from which "concrete"classes `SalariedEmployee`, `HourlyEmployee` and `CommissionEmployee` inherit directly and "concrete" class `BasePlusCommissionEmployee` inherits indirectly. As you'll soon see, *when we invoke each employee's* `earnings` *method off a superclass* `Employee` *reference, the correct earnings subclass calculation is performed,* due to Java's polymorphic capabilities.

Programming in the Specific

Occasionally, when performing polymorphic processing, we need to program "in the specific." Our `Employee` case study demonstrates that a program can determine the type of an object at *execution time* and act on that object accordingly. In the case study, we've decided that `BasePlusCommissionEmployees` should receive 10% raises on their base salaries. So, we use these capabilities to determine whether a particular employee object *is a* `BasePlusCommissionEmployee`. If so, we increase that employee's base salary by 10%.

Interfaces

The chapter continues with an introduction to Java interfaces. An interface describes a set of methods that can be called on an object, but does *not* provide concrete implementations for all the methods. You can declare classes that **implement** (i.e., provide concrete implementations for the methods of) one or more interfaces. Each interface method must be declared in all the classes that explicitly implement the interface. Once a class implements an interface, all objects of that class have an *is-a* relationship with the interface type, and all objects of the class are guaranteed to provide the functionality described by the interface. This is true of all subclasses of that class as well.

Interfaces are particularly useful for assigning common functionality to possibly *unrelated* classes. This allows objects of unrelated classes to be processed polymorphically—objects of classes that implement the same interface can respond to all of the interface method calls. To demonstrate creating and using interfaces, we modify our payroll application to create a general accounts payable application that can calculate payments due for company employees and invoice amounts to be billed for purchased goods. As you'll see, interfaces enable polymorphic capabilities similar to those possible with inheritance.

10.2 Polymorphism Examples

We now consider several additional examples of polymorphism.

Quadrilaterals

If class `Rectangle` is derived from class `Quadrilateral`, then a `Rectangle` object is a more specific version of a `Quadrilateral`. Any operation (e.g., calculating the perimeter or the area) that can be performed on a `Quadrilateral` can also be performed on a `Rectangle`. These operations can also be performed on other `Quadrilaterals`, such as `Squares`, `Parallelograms` and `Trapezoids`. The polymorphism occurs when a program invokes a method through a superclass `Quadrilateral` variable—at execution time, the correct subclass version of the method is called, based on the type of the reference stored in the superclass variable. You'll see a simple code example that illustrates this process in Section 10.3.

Space Objects in a Video Game

Suppose we design a video game that manipulates objects of classes `Martian`, `Venusian`, `Plutonian`, `SpaceShip` and `LaserBeam`. Imagine that each class inherits from the superclass `SpaceObject`, which contains method `draw`. Each subclass implements this method. A screen manager maintains a collection (e.g., a `SpaceObject` array) of references to objects of the various classes. To refresh the screen, the screen manager periodically sends each object the same message—namely, `draw`. However, each object responds its own way, based on its class. For example, a `Martian` object might draw itself in red with green eyes and the appropriate number of antennae. A `SpaceShip` object might draw itself as a bright silver flying saucer. A `LaserBeam` object might draw itself as a bright red beam across the screen. Again, the *same* message (in this case, `draw`) sent to a variety of objects has "many forms" of results.

A screen manager might use polymorphism to facilitate adding new classes to a system with minimal modifications to the system's code. Suppose that we want to add `Mercurian` objects to our video game. To do so, we'd build a class `Mercurian` that extends `SpaceObject` and provides its own `draw` method implementation. When `Mercurian` objects appear in the `SpaceObject` collection, the screen manager code *invokes method* `draw`*, exactly as it does for every other object in the collection, regardless of its type.* So the new `Mercurian` objects simply "plug right in" without any modification of the screen manager code by the programmer. Thus, without modifying the system (other than to build new classes and modify the code that creates new objects), you can use polymorphism to conveniently include additional types that were not envisioned when the system was created.

Software Engineering Observation 10.1

Polymorphism enables you to deal in generalities and let the execution-time environment handle the specifics. You can command objects to behave in manners appropriate to those objects, without knowing their types (as long as the objects belong to the same inheritance hierarchy).

Software Engineering Observation 10.2

Polymorphism promotes extensibility: Software that invokes polymorphic behavior is independent of the object types to which messages are sent. New object types that can respond to existing method calls can be incorporated into a system without modifying the base system. Only client code that instantiates new objects must be modified to accommodate new types.

10.3 Demonstrating Polymorphic Behavior

Section 9.4 created a class hierarchy, in which class `BasePlusCommissionEmployee` inherited from `CommissionEmployee`. The examples in that section manipulated `CommissionEmployee` and `BasePlusCommissionEmployee` objects by using references to them to invoke their methods—we aimed superclass variables at superclass objects and subclass variables at subclass objects. These assignments are natural and straightforward—superclass variables are *intended* to refer to superclass objects, and subclass variables are *intended* to refer to subclass objects. However, as you'll soon see, other assignments are possible.

In the next example, we aim a *superclass* reference at *a subclass* object. We then show how invoking a method on a subclass object via a superclass reference invokes the *subclass* functionality—the type of the *referenced object*, not the type of the *variable*, determines which method is called. This example demonstrates that *an object of a subclass can be treated as an object of its superclass,* enabling various interesting manipulations. A program can create an array of superclass variables that refer to objects of many subclass types. This is allowed because each subclass object *is an* object of its superclass. For instance, we can assign the reference of a `BasePlusCommissionEmployee` object to a superclass `CommissionEmployee` variable, because a `BasePlusCommissionEmployee` *is a* `CommissionEmployee`—we can treat a `BasePlusCommissionEmployee` as a `CommissionEmployee`.

As you'll learn later in the chapter, you *cannot treat a superclass object as a subclass object,* because a superclass object is *not* an object of any of its subclasses. For example, we cannot assign the reference of a `CommissionEmployee` object to a subclass `BasePlusCommissionEmployee` variable, because a `CommissionEmployee` is *not* a `BasePlusCommissionEmployee`—a `CommissionEmployee` does *not* have a `baseSalary` instance variable and does *not* have methods `setBaseSalary` and `getBaseSalary`. The *is-a* relationship applies only *up the hierarchy* from a subclass to its direct (and indirect) superclasses, and *not* vice versa (i.e., not down the hierarchy from a superclass to its subclasses).

The Java compiler *does* allow the assignment of a superclass reference to a subclass variable if we explicitly cast the superclass reference to the subclass type—a technique we discuss in Section 10.5. Why would we ever want to perform such an assignment? A superclass reference can be used to invoke only the methods declared in the superclass—attempting to invoke subclass-only methods through a superclass reference results in compilation errors. If a program needs to perform a subclass-specific operation on a subclass object referenced by a superclass variable, the program must first cast the superclass reference to a subclass reference through a technique known as **downcasting**. This enables the program to invoke subclass methods that are not in the superclass. We show a downcasting example in Section 10.5.

The example in Fig. 10.1 demonstrates three ways to use superclass and subclass variables to store references to superclass and subclass objects. The first two are straightforward—as in Section 9.4, we assign a superclass reference to a superclass variable, and a subclass reference to a subclass variable. Then we demonstrate the relationship between subclasses and superclasses (i.e., the *is-a* relationship) by assigning a subclass reference to a superclass variable. This program uses classes `CommissionEmployee` and `BasePlusCommissionEmployee` from Fig. 9.10 and Fig. 9.11, respectively.

```java
// Fig. 10.1: PolymorphismTest.java
// Assigning superclass and subclass references to superclass and
// subclass variables.

public class PolymorphismTest
{
   public static void main( String[] args )
   {
      // assign superclass reference to superclass variable
      CommissionEmployee commissionEmployee = new CommissionEmployee(
         "Sue", "Jones", "222-22-2222", 10000, .06 );

      // assign subclass reference to subclass variable
      BasePlusCommissionEmployee basePlusCommissionEmployee =
         new BasePlusCommissionEmployee(
         "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );

      // invoke toString on superclass object using superclass variable
      System.out.printf( "%s %s:\n\n%s\n\n",
         "Call CommissionEmployee's toString with superclass reference ",
         "to superclass object", commissionEmployee.toString() );

      // invoke toString on subclass object using subclass variable
      System.out.printf( "%s %s:\n\n%s\n\n",
         "Call BasePlusCommissionEmployee's toString with subclass",
         "reference to subclass object",
         basePlusCommissionEmployee.toString() );

      // invoke toString on subclass object using superclass variable
      CommissionEmployee commissionEmployee2 =
         basePlusCommissionEmployee;
      System.out.printf( "%s %s:\n\n%s\n",
         "Call BasePlusCommissionEmployee's toString with superclass",
         "reference to subclass object", commissionEmployee2.toString() );
   } // end main
} // end class PolymorphismTest
```

```
Call CommissionEmployee's toString with superclass reference to superclass
object:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06
```

Fig. 10.1 | Assigning superclass and subclass references to superclass and subclass variables. (Part 1 of 2.)

```
Call BasePlusCommissionEmployee's toString with subclass reference to
subclass object:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

Call BasePlusCommissionEmployee's toString with superclass reference to
subclass object:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
```

Fig. 10.1 | Assigning superclass and subclass references to superclass and subclass variables. (Part 2 of 2.)

In Fig. 10.1, lines 10–11 create a `CommissionEmployee` object and assign its reference to a `CommissionEmployee` variable. Lines 14–16 create a `BasePlusCommissionEmployee` object and assign its reference to a `BasePlusCommissionEmployee` variable. These assignments are natural—for example, a `CommissionEmployee` variable's primary purpose is to hold a reference to a `CommissionEmployee` object. Lines 19–21 use `commissionEmployee` to invoke `toString` explicitly. Because `commissionEmployee` refers to a `CommissionEmployee` object, superclass `CommissionEmployee`'s version of `toString` is called. Similarly, lines 24–27 use `basePlusCommissionEmployee` to invoke `toString` explicitly on the `BasePlusCommissionEmployee` object. This invokes subclass `BasePlusCommissionEmployee`'s version of `toString`.

Lines 30–31 then assign the reference of subclass object `basePlusCommissionEmployee` to a superclass `CommissionEmployee` variable, which lines 32–34 use to invoke method `toString`. *When a superclass variable contains a reference to a subclass object, and that reference is used to call a method, the subclass version of the method is called.* Hence, `commissionEmployee2.toString()` in line 34 actually calls class `BasePlusCommissionEmployee`'s `toString` method. The Java compiler allows this "crossover" because an object of a subclass *is an* object of its superclass (but not vice versa). When the compiler encounters a method call made through a variable, the compiler determines if the method can be called by checking the variable's class type. If that class contains the proper method declaration (or inherits one), the call is compiled. At execution time, the type of the object to which the variable refers determines the actual method to use. This process, called *dynamic binding*, is discussed in detail in Section 10.5.

10.4 Abstract Classes and Methods

When we think of a class, we assume that programs will create objects of that type. Sometimes it's useful to declare classes—called **abstract classes**—for which you *never* intend to create objects. Because they're used only as superclasses in inheritance hierarchies, we refer to them as **abstract superclasses**. These classes cannot be used to instantiate objects, because, as we'll soon see, abstract classes are *incomplete*. Subclasses must declare the "missing pieces" to become "concrete" classes, from which you can instantiate objects. Otherwise, these subclasses, too, will be abstract. We demonstrate abstract classes in Section 10.5.

Purpose of Abstract Classes

An abstract class's purpose is to provide an appropriate superclass from which other classes can inherit and thus share a common design. In the `Shape` hierarchy of Fig. 9.3, for example, subclasses inherit the notion of what it means to be a `Shape`—perhaps common attributes such as `location`, `color` and `borderThickness`, and behaviors such as `draw`, `move`, `resize` and `changeColor`. Classes that can be used to instantiate objects are called **concrete classes**. Such classes provide implementations of *every* method they declare (some of the implementations can be inherited). For example, we could derive concrete classes `Circle`, `Square` and `Triangle` from abstract superclass `TwoDimensionalShape`. Similarly, we could derive concrete classes `Sphere`, `Cube` and `Tetrahedron` from abstract superclass `ThreeDimensionalShape`. Abstract superclasses are *too general* to create real objects—they specify only what is common among subclasses. We need to be more *specific* before we can create objects. For example, if you send the `draw` message to abstract class `TwoDimensionalShape`, the class knows that two-dimensional shapes should be drawable, but it does not know what specific shape to draw, so it cannot implement a real `draw` method. Concrete classes provide the specifics that make it reasonable to instantiate objects.

Not all hierarchies contain abstract classes. However, you'll often write client code that uses only abstract superclass types to reduce the client code's dependencies on a range of subclass types. For example, you can write a method with a parameter of an abstract superclass type. When called, such a method can receive an object of any concrete class that directly or indirectly extends the superclass specified as the parameter's type.

Abstract classes sometimes constitute several levels of a hierarchy. For example, the `Shape` hierarchy of Fig. 9.3 begins with abstract class `Shape`. On the next level of the hierarchy are *abstract* classes `TwoDimensionalShape` and `ThreeDimensionalShape`. The next level of the hierarchy declares *concrete* classes for `TwoDimensionalShapes` (`Circle`, `Square` and `Triangle`) and for `ThreeDimensionalShapes` (`Sphere`, `Cube` and `Tetrahedron`).

Declaring an Abstract Class and Abstract Methods

You make a class abstract by declaring it with keyword **`abstract`**. An abstract class normally contains one or more **abstract methods**. An abstract method is one with keyword `abstract` in its declaration, as in

```
    public abstract void draw(); // abstract method
```

Abstract methods do *not* provide implementations. A class that contains *any* abstract methods must be explicitly declared `abstract` even if that class contains some concrete (nonabstract) methods. Each concrete subclass of an abstract superclass also must provide concrete implementations of each of the superclass's abstract methods. Constructors and `static` methods cannot be declared `abstract`. Constructors are not inherited, so an `abstract` constructor could never be implemented. Though non-`private static` methods are inherited, they cannot be overridden. Since `abstract` methods are meant to be overridden so that they can process objects based on their types, it would not make sense to declare a `static` method as `abstract`.

Software Engineering Observation 10.3

An abstract class declares common attributes and behaviors (both abstract and concrete) of the various classes in a class hierarchy. An abstract class typically contains one or more abstract methods that subclasses must override if they are to be concrete. The instance variables and concrete methods of an abstract class are subject to the normal rules of inheritance.

Common Programming Error 10.1

Attempting to instantiate an object of an abstract class is a compilation error.

Common Programming Error 10.2

Failure to implement a superclass's abstract methods in a subclass is a compilation error unless the subclass is also declared `abstract`.

Using Abstract Classes to Declare Variables

Although we cannot instantiate objects of abstract superclasses, you'll soon see that we *can* use abstract superclasses to declare variables that can hold references to objects of any concrete class derived from those abstract superclasses. Programs typically use such variables to manipulate subclass objects polymorphically. You also can use abstract superclass names to invoke `static` methods declared in those abstract superclasses.

Consider another application of polymorphism. A drawing program needs to display many shapes, including types of new shapes that you'll add to the system after writing the drawing program. The drawing program might need to display shapes, such as `Circle`s, `Triangle`s, `Rectangle`s or others, that derive from abstract class `Shape`. The drawing program uses `Shape` variables to manage the objects that are displayed. To draw any object in this inheritance hierarchy, the drawing program uses a superclass `Shape` variable containing a reference to the subclass object to invoke the object's `draw` method. This method is declared `abstract` in superclass `Shape`, so each concrete subclass *must* implement method `draw` in a manner specific to that shape—each object in the `Shape` inheritance hierarchy *knows how to draw itself*. The drawing program does not have to worry about the type of each object or whether the program has ever encountered objects of that type.

Layered Software Systems

Polymorphism is particularly effective for implementing so-called layered software systems. In operating systems, for example, each type of physical device could operate quite differently from the others. Even so, commands to read or write data from and to devices may have a certain uniformity. For each device, the operating system uses a piece of software called a *device driver* to control all communication between the system and the device. The write message sent to a device-driver object needs to be interpreted specifically in the context of that driver and how it manipulates devices of a specific type. However, the write call itself really is no different from the write to any other device in the system—place some number of bytes from memory onto that device. An object-oriented operating system might use an abstract superclass to provide an "interface" appropriate for all device drivers. Then, through inheritance from that abstract superclass, subclasses are formed that all behave similarly. The device-driver methods are declared as abstract methods in the abstract superclass. The implementations of these abstract methods are provided in the concrete subclasses that correspond to the specific types of device drivers. New devices are always being developed, often long after the operating system has been released. When you buy a new device, it comes with a device driver provided by the device vendor. The device is immediately operational after you connect it to your computer and install the driver. This is another elegant example of how polymorphism makes systems *extensible*.

10.5 Case Study: Payroll System Using Polymorphism

This section reexamines the `CommissionEmployee`-`BasePlusCommissionEmployee` hierarchy that we explored throughout Section 9.4. Now we use an abstract method and polymorphism to perform payroll calculations based on an enhanced employee inheritance hierarchy that meets the following requirements:

> *A company pays its employees on a weekly basis. The employees are of four types: Salaried employees are paid a fixed weekly salary regardless of the number of hours worked, hourly employees are paid by the hour and receive overtime pay (i.e., 1.5 times their hourly salary rate) for all hours worked in excess of 40 hours, commission employees are paid a percentage of their sales and base-salaried commission employees receive a base salary plus a percentage of their sales. For the current pay period, the company has decided to reward salaried-commission employees by adding 10% to their base salaries. The company wants to write an application that performs its payroll calculations polymorphically.*

We use `abstract` class `Employee` to represent the general concept of an employee. The classes that extend `Employee` are `SalariedEmployee`, `CommissionEmployee` and `HourlyEmployee`. Class `BasePlusCommissionEmployee`—which extends `CommissionEmployee`—represents the last employee type. The UML class diagram in Fig. 10.2 shows the inheritance hierarchy for our polymorphic employee-payroll application. Abstract class name `Employee` is italicized—a convention of the UML.

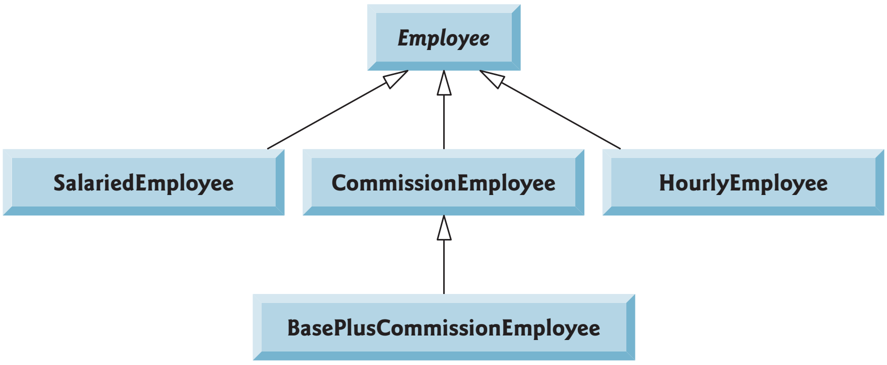


Fig. 10.2 | `Employee` hierarchy UML class diagram.

Abstract superclass `Employee` declares the "interface" to the hierarchy—that is, the set of methods that a program can invoke on all `Employee` objects. We use the term "interface" here in a general sense to refer to the various ways programs can communicate with objects of any `Employee` subclass. Be careful not to confuse the general notion of an “interface” with the formal notion of a Java interface, the subject of Section 10.7. Each employee, regardless of the way his or her earnings are calculated, has a first name, a last name and a social security number, so `private` instance variables `firstName`, `lastName` and `socialSecurityNumber` appear in abstract superclass `Employee`.

The following sections implement the `Employee` class hierarchy of Fig. 10.2. The first section implements abstract superclass `Employee`. The next four sections each implement one of the concrete classes. The last section implements a test program that builds objects of all these classes and processes those objects polymorphically.

10.5.1 Abstract Superclass `Employee`

Class `Employee` (Fig. 10.4) provides methods `earnings` and `toString`, in addition to the *get* and *set* methods that manipulate `Employee`’s instance variables. An `earnings` method certainly applies generically to all employees. But each earnings calculation depends on the employee’s class. So we declare `earnings` as `abstract` in superclass `Employee` because a default implementation does not make sense for that method—there isn’t enough information to determine what amount `earnings` should return. Each subclass overrides `earnings` with an appropriate implementation. To calculate an employee’s earnings, the program assigns to a superclass `Employee` variable a reference to the employee’s object, then invokes the `earnings` method on that variable. We maintain an array of `Employee` variables, each holding a reference to an `Employee` object. (Of course, there cannot be `Employee` objects, because `Employee` is an abstract class. Because of inheritance, however, all objects of all subclasses of `Employee` may nevertheless be thought of as `Employee` objects.) The program will iterate through the array and call method `earnings` for each `Employee` object. Java processes these method calls polymorphically. Declaring `earnings` as an `abstract` method in `Employee` enables the calls to `earnings` through `Employee` variables to compile and forces every direct concrete subclass of `Employee` to override `earnings`.

Method `toString` in class `Employee` returns a `String` containing the first name, last name and social security number of the employee. As we’ll see, each subclass of `Employee` overrides method `toString` to create a `String` representation of an object of that class that contains the employee’s type (e.g., `"salaried employee:"`) followed by the rest of the employee’s information.

The diagram in Fig. 10.3 shows each of the five classes in the hierarchy down the left side and methods `earnings` and `toString` across the top. For each class, the diagram shows the desired results of each method. We do not list superclass `Employee`’s *get* and *set* methods because they’re not overridden in any of the subclasses—each of these methods is inherited and used “as is” by each subclass.

Let’s consider class `Employee`’s declaration (Fig. 10.4). The class includes a constructor that takes the first name, last name and social security number as arguments (lines 11–16); *get* methods that return the first name, last name and social security number (lines 25–28, 37–40 and 49–52, respectively); *set* methods that set the first name, last name and social security number (lines 19–22, 31–34 and 43–46, respectively); method `toString` (lines 55–60), which returns the `String` representation of an `Employee`; and `abstract` method `earnings` (line 63), which will be implemented by each of the concrete subclasses. The `Employee` constructor does not validate its parameters in this example; normally, such validation should be provided.

	earnings	toString
Employee	abstract	*firstName lastName* social security number: *SSN*
Salaried-Employee	weeklySalary	salaried employee: *firstName lastName* social security number: *SSN* weekly salary: *weeklySalary*

Hourly-Employee	if (hours <= 40)
	  wage * hours
	else if (hours > 40)
	{
	  40 * wage +
	  (hours - 40) *
	  wage * 1.5
	}
		hourly employee: *firstName lastName*
		social security number: *SSN*
		hourly wage: *wage*; hours worked: *hours*

Commission-Employee	commissionRate * grossSales	commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*
BasePlus-Commission-Employee	(commissionRate * grossSales) + baseSalary	base salaried commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*; base salary: *baseSalary*

Fig. 10.3 | Polymorphic interface for the `Employee` hierarchy classes.

```java
// Fig. 10.4: Employee.java
// Employee abstract superclass.

public abstract class Employee
{
   private String firstName;
   private String lastName;
   private String socialSecurityNumber;

   // three-argument constructor
   public Employee( String first, String last, String ssn )
   {
      firstName = first;
      lastName = last;
      socialSecurityNumber = ssn;
   } // end three-argument Employee constructor

```

Fig. 10.4 | `Employee` abstract superclass. (Part 1 of 2.)

```
   // set first name
   public void setFirstName( String first )
   {
      firstName = first; // should validate
   } // end method setFirstName

   // return first name
   public String getFirstName()
   {
      return firstName;
   } // end method getFirstName

   // set last name
   public void setLastName( String last )
   {
      lastName = last; // should validate
   } // end method setLastName

   // return last name
   public String getLastName()
   {
      return lastName;
   } // end method getLastName

   // set social security number
   public void setSocialSecurityNumber( String ssn )
   {
      socialSecurityNumber = ssn; // should validate
   } // end method setSocialSecurityNumber

   // return social security number
   public String getSocialSecurityNumber()
   {
      return socialSecurityNumber;
   } // end method getSocialSecurityNumber

   // return String representation of Employee object
   @Override
   public String toString()
   {
      return String.format( "%s %s\nsocial security number: %s",
         getFirstName(), getLastName(), getSocialSecurityNumber() );
   } // end method toString

   // abstract method overridden by concrete subclasses
   public abstract double earnings(); // no implementation here
} // end abstract class Employee
```

Fig. 10.4 | `Employee` abstract superclass. (Part 2 of 2.)

Why did we decide to declare `earnings` as an `abstract` method? It simply does not make sense to provide an implementation of this method in class `Employee`. We cannot calculate the earnings for a *general* `Employee`—we first must know the *specific* type of `Employee` to determine the appropriate earnings calculation. By declaring this method abstract, we indicate that each concrete subclass *must* provide an appropriate earnings implementation and that a program will be able to use superclass Employee variables to invoke method earnings polymorphically for any type of Employee.

10.5.2 Concrete Subclass SalariedEmployee

Class SalariedEmployee (Fig. 10.5) extends class Employee (line 4) and overrides abstract method earnings (lines 33–37), which makes SalariedEmployee a concrete class. The class includes a constructor (lines 9–14) that takes a first name, a last name, a social security number and a weekly salary as arguments; a *set* method to assign a new nonnegative value to instance variable weeklySalary (lines 17–24); a *get* method to return weeklySalary's value (lines 27–30); a method earnings (lines 33–37) to calculate a SalariedEmployee's earnings; and a method toString (lines 40–45), which returns a String including the employee's type, namely, "salaried employee: " followed by employee-specific information produced by superclass Employee's toString method and SalariedEmployee's getWeeklySalary method. Class SalariedEmployee's constructor passes the first name, last name and social security number to the Employee constructor (line 12) to initialize the private instance variables not inherited from the superclass. Method earnings overrides Employee's abstract method earnings to provide a concrete implementation that returns the SalariedEmployee's weekly salary. If we do not implement earnings, class SalariedEmployee must be declared abstract—otherwise, class SalariedEmployee will not compile. Of course, we want SalariedEmployee to be a concrete class in this example.

```
 1  // Fig. 10.5: SalariedEmployee.java
 2  // SalariedEmployee concrete class extends abstract class Employee.
 3
 4  public class SalariedEmployee extends Employee
 5  {
 6     private double weeklySalary;
 7
 8     // four-argument constructor
 9     public SalariedEmployee( String first, String last, String ssn,
10        double salary )
11     {
12        super( first, last, ssn ); // pass to Employee constructor
13        setWeeklySalary( salary ); // validate and store salary
14     } // end four-argument SalariedEmployee constructor
15
16     // set salary
17     public void setWeeklySalary( double salary )
18     {
19        if ( salary >= 0.0 )
20           baseSalary = salary;
21        else
22           throw new IllegalArgumentException(
23              "Weekly salary must be >= 0.0" );
24     } // end method setWeeklySalary
25
```

Fig. 10.5 | SalariedEmployee concrete class extends abstract class Employee. (Part 1 of 2.)

```
26      // return salary
27      public double getWeeklySalary()
28      {
29         return weeklySalary;
30      } // end method getWeeklySalary
31
32      // calculate earnings; override abstract method earnings in Employee
33      @Override
34      public double earnings()
35      {
36         return getWeeklySalary();
37      } // end method earnings
38
39      // return String representation of SalariedEmployee object
40      @Override
41      public String toString()
42      {
43         return String.format( "salaried employee: %s\n%s: $%,.2f",
44            super.toString(), "weekly salary", getWeeklySalary() );
45      } // end method toString
46   } // end class SalariedEmployee
```

Fig. 10.5 | `SalariedEmployee` concrete class extends `abstract` class `Employee`. (Part 2 of 2.)

Method `toString` (lines 40–45) overrides `Employee` method `toString`. If class `SalariedEmployee` did not override `toString`, `SalariedEmployee` would have inherited the `Employee` version of `toString`. In that case, `SalariedEmployee`'s `toString` method would simply return the employee's full name and social security number, which does not adequately represent a `SalariedEmployee`. To produce a complete `String` representation of a `SalariedEmployee`, the subclass's `toString` method returns `"salaried employee: "` followed by the superclass `Employee`-specific information (i.e., first name, last name and social security number) obtained by invoking the superclass's `toString` method (line 44)—this is a nice example of code reuse. The `String` representation of a `SalariedEmployee` also contains the employee's weekly salary obtained by invoking the class's `getWeeklySalary` method.

10.5.3 Concrete Subclass `HourlyEmployee`

Class `HourlyEmployee` (Fig. 10.6) also extends `Employee` (line 4). The class includes a constructor (lines 10–16) that takes as arguments a first name, a last name, a social security number, an hourly wage and the number of hours worked. Lines 19–26 and 35–42 declare *set* methods that assign new values to instance variables `wage` and `hours`, respectively. Method `setWage` (lines 19–26) ensures that `wage` is nonnegative, and method `setHours` (lines 35–42) ensures that `hours` is between 0 and 168 (the total number of hours in a week) inclusive. Class `HourlyEmployee` also includes *get* methods (lines 29–32 and 45–48) to return the values of `wage` and `hours`, respectively; a method `earnings` (lines 51–58) to calculate an `HourlyEmployee`'s earnings; and a method `toString` (lines 61–67), which returns a `String` containing the employee's type (`"hourly employee: "`) and the employee-specific information. The `HourlyEmployee` constructor, like the `SalariedEmployee` constructor, passes the first name, last name and social security number to the superclass `Employee` constructor (line 13) to initialize the `private` instance variables. In addition, method `toString` calls superclass method `toString` (line 65) to obtain the `Employee`-specific information (i.e., first name, last name and social security number)—this is another nice example of code reuse.

```java
 1  // Fig. 10.6: HourlyEmployee.java
 2  // HourlyEmployee class extends Employee.
 3
 4  public class HourlyEmployee extends Employee
 5  {
 6     private double wage; // wage per hour
 7     private double hours; // hours worked for week
 8
 9     // five-argument constructor
10     public HourlyEmployee( String first, String last, String ssn,
11        double hourlyWage, double hoursWorked )
12     {
13        super( first, last, ssn );
14        setWage( hourlyWage ); // validate hourly wage
15        setHours( hoursWorked ); // validate hours worked
16     } // end five-argument HourlyEmployee constructor
17
18     // set wage
19     public void setWage( double hourlyWage )
20     {
21        if ( hourlyWage >= 0.0 )
22           wage = hourlyWage;
23        else
24           throw new IllegalArgumentException(
25              "Hourly wage must be >= 0.0" );
26     } // end method setWage
27
28     // return wage
29     public double getWage()
30     {
31        return wage;
32     } // end method getWage
33
34     // set hours worked
35     public void setHours( double hoursWorked )
36     {
37        if ( ( hoursWorked >= 0.0 ) && ( hoursWorked <= 168.0 ) )
38           hours = hoursWorked;
39        else
40           throw new IllegalArgumentException(
41              "Hours worked must be >= 0.0 and <= 168.0" );
42     } // end method setHours
43
44     // return hours worked
45     public double getHours()
46     {
47        return hours;
48     } // end method getHours
```

Fig. 10.6 | `HourlyEmployee` class extends `Employee`. (Part 1 of 2.)

```java

   // calculate earnings; override abstract method earnings in Employee
   @Override
   public double earnings()
   {
      if ( getHours() <= 40 ) // no overtime
         return getWage() * getHours();
      else
         return 40 * getWage() + ( getHours() - 40 ) * getWage() * 1.5;
   } // end method earnings

   // return String representation of HourlyEmployee object
   @Override
   public String toString()
   {
      return String.format( "hourly employee: %s\n%s: $%,.2f; %s: %,.2f",
         super.toString(), "hourly wage", getWage(),
         "hours worked", getHours() );
   } // end method toString
} // end class HourlyEmployee
```

Fig. 10.6 | HourlyEmployee class extends Employee. (Part 2 of 2.)

10.5.4 Concrete Subclass CommissionEmployee

Class CommissionEmployee (Fig. 10.7) extends class Employee (line 4). The class includes a constructor (lines 10–16) that takes a first name, a last name, a social security number, a sales amount and a commission rate; *set* methods (lines 19–26 and 35–42) to assign new values to instance variables commissionRate and grossSales, respectively; *get* methods (lines 29–32 and 45–48) that retrieve the values of these instance variables; method earnings (lines 51–55) to calculate a CommissionEmployee's earnings; and method toString (lines 58–65), which returns the employee's type, namely, "commission employee: " and employee-specific information. The constructor also passes the first name, last name and social security number to Employee's constructor (line 13) to initialize Employee's private instance variables. Method toString calls superclass method toString (line 62) to obtain the Employee-specific information (i.e., first name, last name and social security number).

```java
 1  // Fig. 10.7: CommissionEmployee.java
 2  // CommissionEmployee class extends Employee.
 3
 4  public class CommissionEmployee extends Employee
 5  {
 6     private double grossSales; // gross weekly sales
 7     private double commissionRate; // commission percentage
 8
 9     // five-argument constructor
10     public CommissionEmployee( String first, String last, String ssn,
11        double sales, double rate )
12     {
13        super( first, last, ssn );
```

Fig. 10.7 | CommissionEmployee class extends Employee. (Part 1 of 2.)

```
      setGrossSales( sales );
      setCommissionRate( rate );
   } // end five-argument CommissionEmployee constructor

   // set commission rate
   public void setCommissionRate( double rate )
   {
      if ( rate > 0.0 && rate < 1.0 )
         commissionRate = rate;
      else
         throw new IllegalArgumentException(
            "Commission rate must be > 0.0 and < 1.0" );
   } // end method setCommissionRate

   // return commission rate
   public double getCommissionRate()
   {
      return commissionRate;
   } // end method getCommissionRate

   // set gross sales amount
   public void setGrossSales( double sales )
   {
      if ( sales >= 0.0 )
         grossSales = sales;
      else
         throw new IllegalArgumentException(
            "Gross sales must be >= 0.0" );
   } // end method setGrossSales

   // return gross sales amount
   public double getGrossSales()
   {
      return grossSales;
   } // end method getGrossSales

   // calculate earnings; override abstract method earnings in Employee
   @Override
   public double earnings()
   {
      return getCommissionRate() * getGrossSales();
   } // end method earnings

   // return String representation of CommissionEmployee object
   @Override
   public String toString()
   {
      return String.format( "%s: %s\n%s: $%,.2f; %s: %.2f",
         "commission employee", super.toString(),
         "gross sales", getGrossSales(),
         "commission rate", getCommissionRate() );
   } // end method toString
} // end class CommissionEmployee
```

Fig. 10.7 | CommissionEmployee class extends Employee. (Part 2 of 2.)

10.5.5 Indirect Concrete Subclass BasePlusCommissionEmployee

Class BasePlusCommissionEmployee (Fig. 10.8) extends class CommissionEmployee (line 4) and therefore is an *indirect* subclass of class Employee. Class BasePlusCommissionEmployee has a constructor (lines 9–14) that takes as arguments a first name, a last name, a social security number, a sales amount, a commission rate and a base salary. It then passes all of these except the base salary to the CommissionEmployee constructor (line 12) to initialize the inherited members. BasePlusCommissionEmployee also contains a *set* method (lines 17–24) to assign a new value to instance variable baseSalary and a *get* method (lines 27–30) to return baseSalary's value. Method earnings (lines 33–37) calculates a BasePlusCommissionEmployee's earnings. Line 36 in method earnings calls superclass CommissionEmployee's earnings method to calculate the commission-based portion of the employee's earnings—this is another nice example of code reuse. BasePlusCommissionEmployee's toString method (lines 40–46) creates a String representation of a BasePlusCommissionEmployee that contains "base-salaried", followed by the String obtained by invoking superclass CommissionEmployee's toString method (another example of code reuse), then the base salary. The result is a String beginning with "base-salaried commission employee" followed by the rest of the BasePlusCommissionEmployee's information. Recall that CommissionEmployee's toString obtains the employee's first name, last name and social security number by invoking the toString method of its superclass (i.e., Employee)—yet another example of code reuse. BasePlusCommissionEmployee's toString initiates a chain of method calls that span all three levels of the Employee hierarchy.

```
 1  // Fig. 10.8: BasePlusCommissionEmployee.java
 2  // BasePlusCommissionEmployee class extends CommissionEmployee.
 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        super( first, last, ssn, sales, rate );
13        setBaseSalary( salary ); // validate and store base salary
14     } // end six-argument BasePlusCommissionEmployee constructor
15
16     // set base salary
17     public void setBaseSalary( double salary )
18     {
19        if ( salary >= 0.0 )
20           baseSalary = salary;
21        else
22           throw new IllegalArgumentException(
23              "Base salary must be >= 0.0" );
24     } // end method setBaseSalary
25
```

Fig. 10.8 | BasePlusCommissionEmployee class extends CommissionEmployee. (Part 1 of 2.)

```
   // return base salary
   public double getBaseSalary()
   {
      return baseSalary;
   } // end method getBaseSalary

   // calculate earnings; override method earnings in CommissionEmployee
   @Override
   public double earnings()
   {
      return getBaseSalary() + super.earnings();
   } // end method earnings

   // return String representation of BasePlusCommissionEmployee object
   @Override
   public String toString()
   {
      return String.format( "%s %s; %s: $%,.2f",
         "base-salaried", super.toString(),
         "base salary", getBaseSalary() );
   } // end method toString
} // end class BasePlusCommissionEmployee
```

Fig. 10.8 | BasePlusCommissionEmployee class extends CommissionEmployee. (Part 2 of 2.)

10.5.6 Polymorphic Processing, Operator instanceof and Downcasting

To test our Employee hierarchy, the application in Fig. 10.9 creates an object of each of the four concrete classes SalariedEmployee, HourlyEmployee, CommissionEmployee and BasePlusCommissionEmployee. The program manipulates these objects nonpolymorphically, via variables of each object's own type, then polymorphically, using an array of Employee variables. While processing the objects polymorphically, the program increases the base salary of each BasePlusCommissionEmployee by 10%—this requires *determining the object's type at execution time*. Finally, the program polymorphically determines and outputs the type of each object in the Employee array. Lines 9–18 create objects of each of the four concrete Employee subclasses. Lines 22–30 output the String representation and earnings of each of these objects *nonpolymorphically*. Each object's toString method is called *implicitly* by printf when the object is output as a String with the %s format specifier.

```
 1  // Fig. 10.9: PayrollSystemTest.java
 2  // Employee hierarchy test program.
 3
 4  public class PayrollSystemTest
 5  {
 6     public static void main( String[] args )
 7     {
 8        // create subclass objects
 9        SalariedEmployee salariedEmployee =
10           new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
```

Fig. 10.9 | Employee hierarchy test program. (Part 1 of 4.)

```
      HourlyEmployee hourlyEmployee =
         new HourlyEmployee( "Karen", "Price", "222-22-2222", 16.75, 40 );
      CommissionEmployee commissionEmployee =
         new CommissionEmployee(
         "Sue", "Jones", "333-33-3333", 10000, .06 );
      BasePlusCommissionEmployee basePlusCommissionEmployee =
         new BasePlusCommissionEmployee(
         "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );

      System.out.println( "Employees processed individually:\n" );

      System.out.printf( "%s\n%s: $%,.2f\n\n",
         salariedEmployee, "earned", salariedEmployee.earnings() );
      System.out.printf( "%s\n%s: $%,.2f\n\n",
         hourlyEmployee, "earned", hourlyEmployee.earnings() );
      System.out.printf( "%s\n%s: $%,.2f\n\n",
         commissionEmployee, "earned", commissionEmployee.earnings() );
      System.out.printf( "%s\n%s: $%,.2f\n\n",
         basePlusCommissionEmployee,
         "earned", basePlusCommissionEmployee.earnings() );

      // create four-element Employee array
      Employee[] employees = new Employee[ 4 ];

      // initialize array with Employees
      employees[ 0 ] = salariedEmployee;
      employees[ 1 ] = hourlyEmployee;
      employees[ 2 ] = commissionEmployee;
      employees[ 3 ] = basePlusCommissionEmployee;

      System.out.println( "Employees processed polymorphically:\n" );

      // generically process each element in array employees
      for ( Employee currentEmployee : employees )
      {
         System.out.println( currentEmployee ); // invokes toString

         // determine whether element is a BasePlusCommissionEmployee
         if ( currentEmployee instanceof BasePlusCommissionEmployee )
         {
            // downcast Employee reference to
            // BasePlusCommissionEmployee reference
            BasePlusCommissionEmployee employee =
               ( BasePlusCommissionEmployee ) currentEmployee;

            employee.setBaseSalary( 1.10 * employee.getBaseSalary() );

            System.out.printf(
               "new base salary with 10%% increase is: $%,.2f\n",
               employee.getBaseSalary() );
         } // end if

```

Fig. 10.9 | Employee hierarchy test program. (Part 2 of 4.)

```java
         System.out.printf(
            "earned $%,.2f\n\n", currentEmployee.earnings() );
      } // end for

      // get type name of each object in employees array
      for ( int j = 0; j < employees.length; j++ )
         System.out.printf( "Employee %d is a %s\n", j,
            employees[ j ].getClass().getName() );
   } // end main
} // end class PayrollSystemTest
```

```
Employees processed individually:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
earned: $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: $16.75; hours worked: 40.00
earned: $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: $10,000.00; commission rate: 0.06
earned: $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: $5,000.00; commission rate: 0.04; base salary: $300.00
earned: $500.00

Employees processed polymorphically:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: $16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: $10,000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: $5,000.00; commission rate: 0.04; base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00
```

Fig. 10.9 | Employee hierarchy test program. (Part 3 of 4.)

```
Employee 0 is a SalariedEmployee
Employee 1 is a HourlyEmployee
Employee 2 is a CommissionEmployee
Employee 3 is a BasePlusCommissionEmployee
```

Fig. 10.9 | `Employee` hierarchy test program. (Part 4 of 4.)

Creating the Array of `Employee`*s*

Line 33 declares `employees` and assigns it an array of four `Employee` variables. Line 36 assigns the reference to a `SalariedEmployee` object to `employees[0]`. Line 37 assigns the reference to an `HourlyEmployee` object to `employees[1]`. Line 38 assigns the reference to a `CommissionEmployee` object to `employees[2]`. Line 39 assigns the reference to a `BasePlusCommissionEmployee` object to `employee[3]`. These assignments are allowed, because a `SalariedEmployee` *is an* `Employee`, an `HourlyEmployee` *is an* `Employee`, a `CommissionEmployee` *is an* `Employee` and a `BasePlusCommissionEmployee` *is an* `Employee`. Therefore, we can assign the references of `SalariedEmployee`, `HourlyEmployee`, `CommissionEmployee` and `BasePlusCommissionEmployee` objects to superclass `Employee` variables, *even though* `Employee` *is an abstract class.*

Polymorphically Processing `Employee`*s*

Lines 44–65 iterate through array `employees` and invoke methods `toString` and `earnings` with `Employee` variable `currentEmployee`, which is assigned the reference to a different `Employee` in the array on each iteration. The output illustrates that the appropriate methods for each class are indeed invoked. All calls to method `toString` and `earnings` are resolved at execution time, based on the type of the object to which `currentEmployee` refers. This process is known as **dynamic binding** or **late binding**. For example, line 46 *implicitly* invokes method `toString` of the object to which `currentEmployee` refers. As a result of dynamic binding, Java decides which class's `toString` method to call *at execution time rather than at compile time*. Only the methods of class `Employee` can be called via an `Employee` variable (and `Employee`, of course, includes the methods of class `Object`). A superclass reference can be used to invoke only methods of the superclass—the subclass method implementations are invoked polymorphically.

Performing Type-Specific Operations on `BasePlusCommissionEmployee`*s*

We perform special processing on `BasePlusCommissionEmployee` objects—as we encounter these objects at execution time, we increase their base salary by 10%. When processing objects polymorphically, we typically do not need to worry about the "specifics," but to adjust the base salary, we *do* have to determine the specific type of `Employee` object at execution time. Line 49 uses the **`instanceof`** operator to determine whether a particular `Employee` object's type is `BasePlusCommissionEmployee`. The condition in line 49 is true if the object referenced by `currentEmployee` *is a* `BasePlusCommissionEmployee`. This would also be true for any object of a `BasePlusCommissionEmployee` subclass because of the *is-a* relationship a subclass has with its superclass. Lines 53–54 downcast `currentEmployee` from type `Employee` to type `BasePlusCommissionEmployee`—this cast is allowed only if the object has an *is-a* relationship with `BasePlusCommissionEmployee`. The condition at line 49 ensures that this is the case. This cast is required if we're to invoke subclass `BasePlusCommissionEmployee` methods `getBaseSalary` and `setBaseSalary` on the current `Employee` object—as you'll see momentarily, *attempting to invoke a subclass-only method directly on a superclass reference is a compilation error.*

Common Programming Error 10.3

Assigning a superclass variable to a subclass variable (without an explicit cast) is a compilation error.

Software Engineering Observation 10.4

If a subclass object's reference has been assigned to a variable of one of its direct or indirect superclasses at execution time, it's acceptable to downcast the reference stored in that superclass variable back to a subclass-type reference. Before performing such a cast, use the `instanceof` *operator to ensure that the object is indeed an object of an appropriate subclass.*

Common Programming Error 10.4

When downcasting a reference, a `ClassCastException` *occurs if the referenced object at execution time does not have an* is-a *relationship with the type specified in the cast operator.*

If the `instanceof` expression in line 49 is `true`, lines 53–60 perform the special processing required for the `BasePlusCommissionEmployee` object. Using `BasePlusCommissionEmployee` variable `employee`, line 56 invokes subclass-only methods `getBaseSalary` and `setBaseSalary` to retrieve and update the employee's base salary with the 10% raise.

Calling `earnings` *Polymorphically*

Lines 63–64 invoke method `earnings` on `currentEmployee`, which polymorphically calls the appropriate subclass object's `earnings` method. Obtaining the earnings of the `SalariedEmployee`, `HourlyEmployee` and `CommissionEmployee` polymorphically in lines 63–64 produces the same results as obtaining these employees' earnings individually in lines 22–27. The earnings amount obtained for the `BasePlusCommissionEmployee` in lines 63–64 is higher than that obtained in lines 28–30, due to the 10% increase in its base salary.

Using Reflection to Get Each `Employee`*'s Class Name*

Lines 68–70 display each employee's type as a `String`, using basic features of Java's so-called reflection capabilities. Every object knows its own class and can access this information through the **`getClass`** method, which all classes inherit from class `Object`. Method `getClass` returns an object of type **`Class`** (from package `java.lang`), which contains information about the object's type, including its class name. Line 70 invokes `getClass` on the current object to get its runtime class. The result of the `getClass` call is used to invoke **`getName`** to get the object's class name.

Avoiding Compilation Errors with Downcasting

In the previous example, we avoided several compilation errors by downcasting an `Employee` variable to a `BasePlusCommissionEmployee` variable in lines 53–54. If you remove the cast operator `(BasePlusCommissionEmployee)` from line 54 and attempt to assign `Employee` variable `currentEmployee` directly to `BasePlusCommissionEmployee` variable `employee`, you'll receive an "`incompatible types`" compilation error. This error indicates that the attempt to assign the reference of superclass object `currentEmployee` to subclass variable `employee` is not allowed. The compiler prevents this assignment because a `CommissionEmployee` is not a `BasePlusCommissionEmployee`—*the* is-a *relationship applies only between the subclass and its superclasses, not vice versa.*

Similarly, if lines 56 and 60 used superclass variable `currentEmployee` to invoke subclass-only methods `getBaseSalary` and `setBaseSalary`, we'd receive "`cannot find symbol`" compilation errors at these lines. Attempting to invoke subclass-only methods via a superclass variable is not allowed—even though lines 56 and 60 execute only if `instanceof` in line 49 returns `true` to indicate that `currentEmployee` holds a reference to a `BasePlusCommissionEmployee` object. Using a superclass `Employee` variable, we can invoke only methods found in class `Employee`—`earnings`, `toString` and `Employee`'s *get* and *set* methods.

Software Engineering Observation 10.5

Although the actual method that's called depends on the runtime type of the object to which a variable refers, a variable can be used to invoke only those methods that are members of that variable's type, which the compiler verifies.

10.5.7 Summary of the Allowed Assignments Between Superclass and Subclass Variables

Now that you've seen a complete application that processes diverse subclass objects polymorphically, we summarize what you can and cannot do with superclass and subclass objects and variables. Although a subclass object also *is a* superclass object, the two objects are nevertheless different. As discussed previously, subclass objects can be treated as objects of their superclass. But because the subclass can have additional subclass-only members, assigning a superclass reference to a subclass variable is not allowed without an explicit cast—such an assignment would leave the subclass members undefined for the superclass object.

We've discussed four ways to assign superclass and subclass references to variables of superclass and subclass types:

1. Assigning a superclass reference to a superclass variable is straightforward.
2. Assigning a subclass reference to a subclass variable is straightforward.
3. Assigning a subclass reference to a superclass variable is safe, because the subclass object *is an* object of its superclass. However, the superclass variable can be used to refer *only* to superclass members. If this code refers to subclass-only members through the superclass variable, the compiler reports errors.
4. Attempting to assign a superclass reference to a subclass variable is a compilation error. To avoid this error, the superclass reference must be cast to a subclass type explicitly. At *execution time*, if the object to which the reference refers is *not* a subclass object, an exception will occur. (For more on exception handling, see Chapter 11.) You should use the `instanceof` operator to ensure that such a cast is performed only if the object is a subclass object.

10.6 `final` Methods and Classes

We saw in Sections 6.3 and 6.10 that variables can be declared `final` to indicate that they cannot be modified after they're initialized—such variables represent constant values. It's also possible to declare methods, method parameters and classes with the `final` modifier.

Final Methods Cannot Be Overridden

A **`final` method** in a superclass *cannot* be overridden in a subclass—this guarantees that the `final` method implementation will be used by all direct and indirect subclasses in the hierarchy. Methods that are declared `private` are implicitly `final`, because it's not possible to override them in a subclass. Methods that are declared `static` are also implicitly `final`. A `final` method's declaration can never change, so all subclasses use the same method implementation, and calls to `final` methods are resolved at compile time—this is known as **static binding**.

Final Classes Cannot Be Superclasses

A **`final` class** that's declared `final` cannot be a superclass (i.e., a class cannot extend a `final` class). All methods in a `final` class are implicitly `final`. Class `String` is an example of a `final` class. If you were allowed to create a subclass of `String`, objects of that subclass could be used wherever `String`s are expected. Since class `String` cannot be extended, programs that use `String`s can rely on the functionality of `String` objects as specified in the Java API. Making the class `final` also prevents programmers from creating subclasses that might bypass security restrictions. For more insights on the use of keyword `final`, visit

```
download.oracle.com/javase/tutorial/java/IandI/final.html
```

and

```
www.ibm.com/developerworks/java/library/j-jtp1029.html
```

Common Programming Error 10.5

Attempting to declare a subclass of a `final` class is a compilation error.

Software Engineering Observation 10.6

In the Java API, the vast majority of classes are not *declared `final`. This enables inheritance and polymorphism. However, in some cases, it's important to declare classes `final`—typically for security reasons.*

10.7 Case Study: Creating and Using Interfaces

Our next example (Figs. 10.11–10.15) reexamines the payroll system of Section 10.5. Suppose that the company involved wishes to perform several accounting operations in a single accounts payable application—in addition to calculating the earnings that must be paid to each employee, the company must also calculate the payment due on each of several invoices (i.e., bills for goods purchased). Though applied to unrelated things (i.e., employees and invoices), both operations have to do with obtaining some kind of payment amount. For an employee, the payment refers to the employee's earnings. For an invoice, the payment refers to the total cost of the goods listed on the invoice. Can we calculate such *different* things as the payments due for employees and invoices in *a single* application polymorphically? Does Java offer a capability requiring that *unrelated* classes implement a set of *common* methods (e.g., a method that calculates a payment amount)? Java **interfaces** offer exactly this capability.

Standardizing Interactions

Interfaces define and standardize the ways in which things such as people and systems can interact with one another. For example, the controls on a radio serve as an interface between radio users and a radio's internal components. The controls allow users to perform only a limited set of operations (e.g., change the station, adjust the volume, choose between AM and FM), and different radios may implement the controls in different ways (e.g., using push buttons, dials, voice commands). The interface specifies *what* operations a radio must permit users to perform but does not specify *how* the operations are performed.

Software Objects Communicate Via Interfaces

Software objects also communicate via interfaces. A Java interface describes a set of methods that can be called on an object to tell it, for example, to perform some task or return some piece of information. The next example introduces an interface named `Payable` to describe the functionality of any object that must be capable of being paid and thus must offer a method to determine the proper payment amount due. An **interface declaration** begins with the keyword **`interface`** and contains only constants and `abstract` methods. Unlike classes, all interface members must be `public`, and *interfaces may not specify any implementation details*, such as concrete method declarations and instance variables. All methods declared in an interface are implicitly `public abstract` methods, and all fields are implicitly `public`, `static` and `final`. [*Note:* As of Java SE 5, it became a better programming practice to declare sets of constants as enumerations with keyword `enum`. See Section 6.10 for an introduction to `enum` and Section 8.9 for additional `enum` details.]

Good Programming Practice 10.1

According to Chapter 9 of the Java Language Specification*, it's proper style to declare an interface's methods without keywords* `public` *and* `abstract`*, because they're redundant in interface method declarations. Similarly, constants should be declared without keywords* `public`*,* `static` *and* `final`*, because they, too, are redundant.*

Using an Interface

To use an interface, a concrete class must specify that it **`implements`** the interface and must declare each method in the interface with the signature specified in the interface declaration. To specify that a class implements an interface add the `implements` keyword and the name of the interface to the end of your class declaration's first line. A class that does not implement *all* the methods of the interface is an *abstract* class and must be declared `abstract`. Implementing an interface is like signing a *contract* with the compiler that states, "I will declare all the methods specified by the interface or I will declare my class `abstract`."

Common Programming Error 10.6

Failing to implement any method of an interface in a concrete class that `implements` *the interface results in a compilation error indicating that the class must be declared* `abstract`*.*

Relating Disparate Types

An interface is often used when disparate (i.e., unrelated) classes need to share common methods and constants. This allows objects of unrelated classes to be processed polymorphically—objects of classes that implement the same interface can respond to the same method calls. You can create an interface that describes the desired functionality, then implement this interface in any classes that require that functionality. For example, in the accounts payable application developed in this section, we implement interface `Payable` in any class that must be able to calculate a payment amount (e.g., `Employee`, `Invoice`).

Interfaces vs. Abstract Classes

An interface is often used in place of an `abstract` class when there's no default implementation to inherit—that is, no fields and no default method implementations. Like `public abstract` classes, interfaces are typically `public` types. Like a `public` class, a `public` interface must be declared in a file with the same name as the interface and the `.java` file-name extension.

Tagging Interfaces

We'll see in Chapter 17, Files, Streams and Object Serialization, the notion of "tagging interfaces"—empty interfaces that have *no* methods or constant values. They're used to add *is-a* relationships to classes. For example, in Chapter 17 we'll discuss a mechanism called object serialization, which can convert objects to byte representations and can convert those byte representations back to objects. To enable this mechanism to work with your objects, you simply have to mark them as `Serializable` by adding `implements Serializable` to the end of your class declaration's first line. Then, all the objects of your class have the *is-a* relationship with `Serializable`.

10.7.1 Developing a `Payable` Hierarchy

To build an application that can determine payments for employees and invoices alike, we first create interface `Payable`, which contains method `getPaymentAmount` that returns a `double` amount that must be paid for an object of any class that implements the interface. Method `getPaymentAmount` is a general-purpose version of method `earnings` of the `Employee` hierarchy—method `earnings` calculates a payment amount specifically for an `Employee`, while `getPaymentAmount` can be applied to a broad range of unrelated objects. After declaring interface `Payable`, we introduce class `Invoice`, which `implements` interface `Payable`. We then modify class `Employee` such that it also implements interface `Payable`. Finally, we update `Employee` subclass `SalariedEmployee` to "fit" into the `Payable` hierarchy by renaming `SalariedEmployee` method `earnings` as `getPaymentAmount`.

Good Programming Practice 10.2

When declaring a method in an interface, choose a method name that describes the method's purpose in a general *manner, because the method may be implemented by many unrelated classes.*

Classes `Invoice` and `Employee` both represent things for which the company must be able to calculate a payment amount. Both classes implement the `Payable` interface, so a program can invoke method `getPaymentAmount` on `Invoice` objects and `Employee` objects alike. As we'll soon see, this enables the polymorphic processing of `Invoices` and `Employees` required for the company's accounts payable application.

The UML class diagram in Fig. 10.10 shows the hierarchy used in our accounts payable application. The hierarchy begins with interface `Payable`. The UML distinguishes an interface from other classes by placing the word "interface" in guillemets (« and ») above the interface name. The UML expresses the relationship between a class and an interface through a relationship known as **realization**. A class is said to "realize," or implement, the methods of an interface. A class diagram models a realization as a dashed arrow with a hollow arrowhead pointing from the implementing class to the interface. The diagram in Fig. 10.10 indicates that classes `Invoice` and `Employee` each realize (i.e., implement) interface Payable. As in the class diagram of Fig. 10.2, class Employee appears in italics, indicating that it's an abstract class. Concrete class SalariedEmployee extends Employee and *inherits its superclass's realization relationship* with interface Payable.

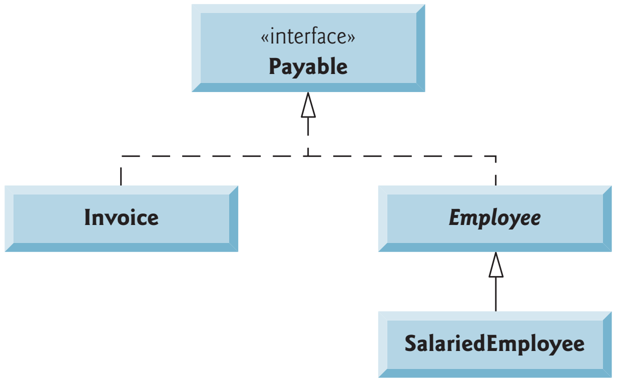


Fig. 10.10 | Payable interface hierarchy UML class diagram.

10.7.2 Interface Payable

The declaration of interface Payable begins in Fig. 10.11 at line 4. Interface Payable contains public abstract method getPaymentAmount (line 6). The method is not explicitly declared public or abstract. Interface methods are always public and abstract, so they do not need to be declared as such. Interface Payable has only one method—interfaces can have any number of methods. In addition, method getPaymentAmount has no parameters, but interface methods *can* have parameters. Interfaces may also contain fields that are implicitly final and static.

```
1  // Fig. 10.11: Payable.java
2  // Payable interface declaration.
3
4  public interface Payable
5  {
6     double getPaymentAmount(); // calculate payment; no implementation
7  } // end interface Payable
```

Fig. 10.11 | Payable interface declaration.

10.7.3 Class Invoice

We now create class Invoice (Fig. 10.12) to represent a simple invoice that contains billing information for only one kind of part. The class declares private instance variables partNumber, partDescription, quantity and pricePerItem (in lines 6–9) that indicate the part number, a description of the part, the quantity of the part ordered and the price per item. Class Invoice also contains a constructor (lines 12–19), *get* and *set* methods (lines 22–74) that manipulate the class's instance variables and a toString method (lines 77–83) that returns a String representation of an Invoice object. Methods setQuantity (lines 46–52) and setPricePerItem (lines 61–68) ensure that quantity and pricePerItem obtain only nonnegative values.

```java
// Fig. 10.12: Invoice.java
// Invoice class that implements Payable.

public class Invoice implements Payable
{
   private String partNumber;
   private String partDescription;
   private int quantity;
   private double pricePerItem;

   // four-argument constructor
   public Invoice( String part, String description, int count,
      double price )
   {
      partNumber = part;
      partDescription = description;
      setQuantity( count ); // validate and store quantity
      setPricePerItem( price ); // validate and store price per item
   } // end four-argument Invoice constructor

   // set part number
   public void setPartNumber( String part )
   {
      partNumber = part; // should validate
   } // end method setPartNumber

   // get part number
   public String getPartNumber()
   {
      return partNumber;
   } // end method getPartNumber

   // set description
   public void setPartDescription( String description )
   {
      partDescription = description; // should validate
   } // end method setPartDescription

   // get description
   public String getPartDescription()
   {
      return partDescription;
   } // end method getPartDescription

   // set quantity
   public void setQuantity( int count )
   {
      if ( count >= 0 )
         quantity = count;
      else
         throw new IllegalArgumentException( "Quantity must be >= 0" );
   } // end method setQuantity

```

Fig. 10.12 | Invoice class that implements Payable. (Part 1 of 2.)

```java
54     // get quantity
55     public int getQuantity()
56     {
57        return quantity;
58     } // end method getQuantity
59
60     // set price per item
61     public void setPricePerItem( double price )
62     {
63        if ( price >= 0.0 )
64           pricePerItem = price;
65        else
66           throw new IllegalArgumentException(
67              "Price per item must be >= 0" );
68     } // end method setPricePerItem
69
70     // get price per item
71     public double getPricePerItem()
72     {
73        return pricePerItem;
74     } // end method getPricePerItem
75
76     // return String representation of Invoice object
77     @Override
78     public String toString()
79     {
80        return String.format( "%s: \n%s: %s (%s) \n%s: %d \n%s: $%,.2f",
81           "invoice", "part number", getPartNumber(), getPartDescription(),
82           "quantity", getQuantity(), "price per item", getPricePerItem() );
83     } // end method toString
84
85     // method required to carry out contract with interface Payable
86     @Override
87     public double getPaymentAmount()
88     {
89        return getQuantity() * getPricePerItem(); // calculate total cost
90     } // end method getPaymentAmount
91  } // end class Invoice
```

Fig. 10.12 | `Invoice` class that implements `Payable`. (Part 2 of 2.)

Line 4 indicates that class `Invoice` implements interface `Payable`. Like all classes, class `Invoice` also implicitly extends `Object`. Java does not allow subclasses to inherit from more than one superclass, but it allows a class to inherit from one superclass and implement as many interfaces as it needs. To implement more than one interface, use a comma-separated list of interface names after keyword `implements` in the class declaration, as in:

```
public class ClassName extends SuperclassName implements FirstInterface,
   SecondInterface, …
```

Software Engineering Observation 10.7

All objects of a class that implement multiple interfaces have the is-a *relationship with each implemented interface type.*

Class `Invoice` implements the one method in interface `Payable`—method `getPaymentAmount` is declared in lines 86–90. The method calculates the total payment required to pay the invoice. The method multiplies the values of `quantity` and `pricePerItem` (obtained through the appropriate *get* methods) and returns the result (line 89). This method satisfies the implementation requirement for this method in interface `Payable`—we've fulfilled the interface contract with the compiler.

10.7.4 Modifying Class `Employee` to Implement Interface `Payable`

We now modify class `Employee` such that it implements interface `Payable`. Figure 10.13 contains the modified class, which is identical to that of Fig. 10.4 with two exceptions. First, line 4 of Fig. 10.13 indicates that class `Employee` now `implements` interface `Payable`. So we must rename `earnings` to `getPaymentAmount` throughout the `Employee` hierarchy. As with method `earnings` in the version of class `Employee` in Fig. 10.4, however, it does not make sense to implement method `getPaymentAmount` in class `Employee` because we cannot calculate the earnings payment owed to a general `Employee`—we must first know the specific type of `Employee`. In Fig. 10.4, we declared method `earnings` as `abstract` for this reason, so class `Employee` had to be declared `abstract`. This forced each `Employee` concrete subclass to override `earnings` with an implementation.

```
 1  // Fig. 10.13: Employee.java
 2  // Employee abstract superclass that implements Payable.
 3
 4  public abstract class Employee implements Payable
 5  {
 6     private String firstName;
 7     private String lastName;
 8     private String socialSecurityNumber;
 9
10     // three-argument constructor
11     public Employee( String first, String last, String ssn )
12     {
13        firstName = first;
14        lastName = last;
15        socialSecurityNumber = ssn;
16     } // end three-argument Employee constructor
17
18     // set first name
19     public void setFirstName( String first )
20     {
21        firstName = first; // should validate
22     } // end method setFirstName
23
24     // return first name
25     public String getFirstName()
26     {
27        return firstName;
28     } // end method getFirstName
29
```

Fig. 10.13 | `Employee` class that implements `Payable`. (Part 1 of 2.)

```
30     // set last name
31     public void setLastName( String last )
32     {
33        lastName = last; // should validate
34     } // end method setLastName
35
36     // return last name
37     public String getLastName()
38     {
39        return lastName;
40     } // end method getLastName
41
42     // set social security number
43     public void setSocialSecurityNumber( String ssn )
44     {
45        socialSecurityNumber = ssn; // should validate
46     } // end method setSocialSecurityNumber
47
48     // return social security number
49     public String getSocialSecurityNumber()
50     {
51        return socialSecurityNumber;
52     } // end method getSocialSecurityNumber
53
54     // return String representation of Employee object
55     @Override
56     public String toString()
57     {
58        return String.format( "%s %s\nsocial security number: %s",
59           getFirstName(), getLastName(), getSocialSecurityNumber() );
60     } // end method toString
61
62     // Note: We do not implement Payable method getPaymentAmount here so
63     // this class must be declared abstract to avoid a compilation error.
64  } // end abstract class Employee
```

Fig. 10.13 | `Employee` class that implements `Payable`. (Part 2 of 2.)

In Fig. 10.13, we handle this situation differently. Recall that when a class implements an interface, it makes a *contract* with the compiler stating either that the class will implement *each* of the methods in the interface or that the class will be declared `abstract`. If the latter option is chosen, we do not need to declare the interface methods as `abstract` in the `abstract` class—they're already implicitly declared as such in the interface. Any concrete subclass of the `abstract` class must implement the interface methods to fulfill the superclass's contract with the compiler. If the subclass does not do so, it too must be declared `abstract`. As indicated by the comments in lines 62–63, class `Employee` of Fig. 10.13 does *not* implement method `getPaymentAmount`, so the class is declared `abstract`. Each direct `Employee` subclass *inherits the superclass's contract* to implement method `getPaymentAmount` and thus must implement this method to become a concrete class for which objects can be instantiated. A class that extends one of `Employee`'s concrete subclasses will inherit an implementation of `getPaymentAmount` and thus will also be a concrete class.

10.7.5 Modifying Class SalariedEmployee for Use in the Payable Hierarchy

Figure 10.14 contains a modified SalariedEmployee class that extends Employee and fulfills superclass Employee's contract to implement Payable method getPaymentAmount. This version of SalariedEmployee is identical to that of Fig. 10.5, but it replaces method earnings with method getPaymentAmount (lines 34–38). Recall that the Payable version of the method has a more *general* name to be applicable to possibly *disparate* classes. The remaining Employee subclasses (e.g., HourlyEmployee, CommissionEmployee and BasePlusCommissionEmployee) also must be modified to contain method getPaymentAmount in place of earnings to reflect the fact that Employee now implements Payable. We leave these modifications as an exercise (Exercise 10.11) and use only SalariedEmployee in our test program here. Exercise 10.12 asks you to implement interface Payable in the entire Employee class hierarchy of Figs. 10.4–10.9 without modifying the Employee subclasses.

When a class implements an interface, the same *is-a* relationship provided by inheritance applies. Class Employee implements Payable, so we can say that an Employee *is a* Payable. In fact, objects of any classes that extend Employee are also Payable objects. SalariedEmployee objects, for instance, are Payable objects. Objects of any subclasses of the class that implements the interface can also be thought of as objects of the interface type. Thus, just as we can assign the reference of a SalariedEmployee object to a superclass Employee variable, we can assign the reference of a SalariedEmployee object to an interface Payable variable. Invoice implements Payable, so an Invoice object also *is a* Payable object, and we can assign the reference of an Invoice object to a Payable variable.

Software Engineering Observation 10.8

When a method parameter is declared with a superclass or interface type, the method processes the object received as an argument polymorphically.

Software Engineering Observation 10.9

Using a superclass reference, we can polymorphically invoke any method declared in the superclass and its superclasses (e.g., class Object*). Using an interface reference, we can polymorphically invoke any method declared in the interface, its superinterfaces (one interface can extend another) and in class* Object*—a variable of an interface type must refer to an object to call methods, and all objects have the methods of class* Object*.*

```
 1  // Fig. 10.14: SalariedEmployee.java
 2  // SalariedEmployee class extends Employee, which implements Payable.
 3
 4  public class SalariedEmployee extends Employee
 5  {
 6     private double weeklySalary;
 7
 8     // four-argument constructor
 9     public SalariedEmployee( String first, String last, String ssn,
10        double salary )
11     {
```

Fig. 10.14 | SalariedEmployee class that implements interface Payable method getPaymentAmount. (Part 1 of 2.)

```java
      super( first, last, ssn ); // pass to Employee constructor
      setWeeklySalary( salary ); // validate and store salary
   } // end four-argument SalariedEmployee constructor

   // set salary
   public void setWeeklySalary( double salary )
   {
      if ( salary >= 0.0 )
         baseSalary = salary;
      else
         throw new IllegalArgumentException(
            "Weekly salary must be >= 0.0" );
   } // end method setWeeklySalary

   // return salary
   public double getWeeklySalary()
   {
      return weeklySalary;
   } // end method getWeeklySalary

   // calculate earnings; implement interface Payable method that was
   // abstract in superclass Employee
   @Override
   public double getPaymentAmount()
   {
      return getWeeklySalary();
   } // end method getPaymentAmount

   // return String representation of SalariedEmployee object
   @Override
   public String toString()
   {
      return String.format( "salaried employee: %s\n%s: $%,.2f",
         super.toString(), "weekly salary", getWeeklySalary() );
   } // end method toString
} // end class SalariedEmployee
```

Fig. 10.14 | `SalariedEmployee` class that implements interface `Payable` method `getPaymentAmount`. (Part 2 of 2.)

10.7.6 Using Interface `Payable` to Process `Invoice`s and `Employee`s Polymorphically

`PayableInterfaceTest` (Fig. 10.15) illustrates that interface `Payable` can be used to process a set of `Invoice`s and `Employee`s polymorphically in a single application. Line 9 declares `payableObjects` and assigns it an array of four `Payable` variables. Lines 12–13 assign the references of `Invoice` objects to the first two elements of `payableObjects`. Lines 14–17 then assign the references of `SalariedEmployee` objects to the remaining two elements of `payableObjects`. These assignments are allowed because an `Invoice` *is a* `Payable`, a `SalariedEmployee` *is an* `Employee` and an `Employee` *is a* `Payable`. Lines 23–29 use the enhanced `for` statement to polymorphically process each `Payable` object in `payableObjects`, printing the object as a `String`, along with the payment amount due. Line 27 invokes method `toString` via a `Payable` interface reference, even though `toString` is not declared in interface `Payable`—*all references (including those of interface types) refer to objects that extend `Object` and therefore have a `toString` method.* (Method `toString` also can be invoked *implicitly* here.) Line 28 invokes `Payable` method `getPaymentAmount` to obtain the payment amount for each object in `payableObjects`, regardless of the actual type of the object. The output reveals that the method calls in lines 27–28 invoke the appropriate class's implementation of methods `toString` and `getPaymentAmount`. For instance, when `currentPayable` refers to an `Invoice` during the first iteration of the `for` loop, class `Invoice`'s `toString` and `getPaymentAmount` execute.

```
 1  // Fig. 10.15: PayableInterfaceTest.java
 2  // Tests interface Payable.
 3
 4  public class PayableInterfaceTest
 5  {
 6     public static void main( String[] args )
 7     {
 8        // create four-element Payable array
 9        Payable[] payableObjects = new Payable[ 4 ];
10
11        // populate array with objects that implement Payable
12        payableObjects[ 0 ] = new Invoice( "01234", "seat", 2, 375.00 );
13        payableObjects[ 1 ] = new Invoice( "56789", "tire", 4, 79.95 );
14        payableObjects[ 2 ] =
15           new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
16        payableObjects[ 3 ] =
17           new SalariedEmployee( "Lisa", "Barnes", "888-88-8888", 1200.00 );
18
19        System.out.println(
20           "Invoices and Employees processed polymorphically:\n" );
21
22        // generically process each element in array payableObjects
23        for ( Payable currentPayable : payableObjects )
24        {
25           // output currentPayable and its appropriate payment amount
26           System.out.printf( "%s \n%s: $%,.2f\n\n",
27              currentPayable.toString(),
28              "payment due", currentPayable.getPaymentAmount() );
29        } // end for
30     } // end main
31  } // end class PayableInterfaceTest
```

```
Invoices and Employees processed polymorphically:

invoice:
part number: 01234 (seat)
quantity: 2
price per item: $375.00
payment due: $750.00
```

Fig. 10.15 | `Payable` interface test program processing `Invoice`s and `Employee`s polymorphically. (Part 1 of 2.)

```
invoice:
part number: 56789 (tire)
quantity: 4
price per item: $79.95
payment due: $319.80

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
payment due: $800.00

salaried employee: Lisa Barnes
social security number: 888-88-8888
weekly salary: $1,200.00
payment due: $1,200.00
```

Fig. 10.15 | `Payable` interface test program processing `Invoices` and `Employees` polymorphically. (Part 2 of 2.)

10.7.7 Common Interfaces of the Java API

In this section, we overview several common interfaces found in the Java API. The power and flexibility of interfaces is used frequently throughout the Java API. These interfaces are implemented and used in the same manner as the interfaces you create (e.g., interface `Payable` in Section 10.7.2). The Java API's interfaces enable you to use your own classes within the frameworks provided by Java, such as comparing objects of your own types and creating tasks that can execute concurrently with other tasks in the same program. Figure 10.16 overviews a few of the more popular interfaces of the Java API that we use in *Java How to Program, Ninth Edition*.

Interface	Description
`Comparable`	Java contains several comparison operators (e.g., <, <=, >, >=, ==, !=) that allow you to compare primitive values. However, these operators *cannot* be used to compare objects. Interface `Comparable` is used to allow objects of a class that `implements` the interface to be compared to one another. Interface `Comparable` is commonly used for ordering objects in a collection such as an array. We use `Comparable` in Chapter 20, Generic Collections, and Chapter 21, Generic Classes and Methods.
`Serializable`	An interface used to identify classes whose objects can be written to (i.e., serialized) or read from (i.e., deserialized) some type of storage (e.g., file on disk, database field) or transmitted across a network. We use `Serializable` in Chapter 17, Files, Streams and Object Serialization, and Chapter 27, Networking.
`Runnable`	Implemented by any class for which objects of that class should be able to execute in parallel using a technique called multithreading (discussed in Chapter 26, Multithreading). The interface contains one method, `run`, which describes the behavior of an object when executed.

Fig. 10.16 | Common interfaces of the Java API. (Part 1 of 2.)

Interface	Description
GUI event-listener interfaces	You work with graphical user interfaces (GUIs) every day. In your web browser, you might type the address of a website to visit, or you might click a button to return to a previous site. The browser responds to your interaction and performs the desired task. Your interaction is known as an event, and the code that the browser uses to respond to an event is known as an event handler. In Chapter 14, GUI Components: Part 1, and Chapter 25, GUI Components: Part 2, you'll learn how to build GUIs and event handlers that respond to user interactions. Event handlers are declared in classes that implement an appropriate event-listener interface. Each event-listener interface specifies one or more methods that must be implemented to respond to user interactions.
`SwingConstants`	Contains a set of constants used in GUI programming to position GUI elements on the screen. We explore GUI programming in Chapters 14 and 25.

Fig. 10.16 | Common interfaces of the Java API. (Part 2 of 2.)

10.8 (Optional) GUI and Graphics Case Study: Drawing with Polymorphism

You may have noticed in the drawing program created in GUI and Graphics Case Study Exercise 8.1 (and modified in GUI and Graphics Case Study Exercise 9.1) that shape classes have many similarities. Using inheritance, we can "factor out" the common features from all three classes and place them in a single shape superclass. Then, using variables of the superclass type, we can manipulate shape objects polymorphically. Removing the redundant code will result in a smaller, more flexible program that's easier to maintain.

GUI and Graphics Case Study Exercises

10.1 Modify the `MyLine`, `MyOval` and `MyRectangle` classes of GUI and Graphics Case Study Exercise 8.1 and Exercise 9.1 to create the class hierarchy in Fig. 10.17. Classes of the `MyShape` hierarchy should be "smart" shape classes that know how to draw themselves (if provided with a `Graphics` object that tells them where to draw). Once the program creates an object from this hierarchy, it can manipulate it polymorphically for the rest of its lifetime as a `MyShape`.

In your solution, class `MyShape` in Fig. 10.17 *must* be `abstract`. Since `MyShape` represents any shape in general, you cannot implement a `draw` method without knowing exactly what shape it is. The data representing the coordinates and color of the shapes in the hierarchy should be declared as `private` members of class `MyShape`. In addition to the common data, class `MyShape` should declare the following methods:

a) A no-argument constructor that sets all the coordinates of the shape to `0` and the color to `Color.BLACK`.

b) A constructor that initializes the coordinates and color to the values of the arguments supplied.

c) *Set* methods for the individual coordinates and color that allow the programmer to set any piece of data independently for a shape in the hierarchy.

d) *Get* methods for the individual coordinates and color that allow the programmer to retrieve any piece of data independently for a shape in the hierarchy.

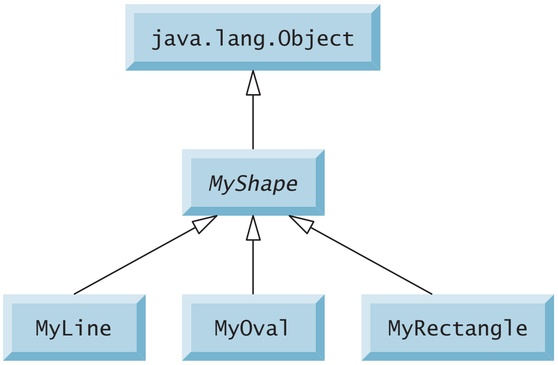


Fig. 10.17 | MyShape hierarchy.

e) The abstract method

```
public abstract void draw( Graphics g );
```

which the program's paintComponent method will call to draw a shape on the screen.

To ensure proper encapsulation, all data in class MyShape must be private. This requires declaring proper *set* and *get* methods to manipulate the data. Class MyLine should provide a no-argument constructor and a constructor with arguments for the coordinates and color. Classes MyOval and MyRectangle should provide a no-argument constructor and a constructor with arguments for the coordinates, color and determining whether the shape is filled. The no-argument constructor should, in addition to setting the default values, set the shape to be an unfilled shape.

You can draw lines, rectangles and ovals if you know two points in space. Lines require *x1*, *y1*, *x2* and *y2* coordinates. The drawLine method of the Graphics class will connect the two points supplied with a line. If you have the same four coordinate values (*x1*, *y1*, *x2* and *y2*) for ovals and rectangles, you can calculate the four arguments needed to draw them. Each requires an upper-left *x*-coordinate value (the smaller of the two *x*-coordinate values), an upper-left *y*-coordinate value (the smaller of the two *y*-coordinate values), a *width* (the absolute value of the difference between the two *x*-coordinate values) and a *height* (the absolute value of the difference between the two *y*-coordinate values). Rectangles and ovals should also have a filled flag that determines whether to draw the shape as a filled shape.

There should be no MyLine, MyOval or MyRectangle variables in the program—only MyShape variables that contain references to MyLine, MyOval and MyRectangle objects. The program should generate random shapes and store them in an array of type MyShape. Method paintComponent should walk through the MyShape array and draw every shape (i.e., polymorphically calling every shape's draw method).

Allow the user to specify (via an input dialog) the number of shapes to generate. The program will then generate and display the shapes along with a status bar that informs the user how many of each shape were created.

10.2 *(Drawing Application Modification)* In Exercise 10.1, you created a MyShape hierarchy in which classes MyLine, MyOval and MyRectangle extend MyShape directly. If your hierarchy was properly designed, you should be able to see the similarities between the MyOval and MyRectangle classes. Redesign and reimplement the code for the MyOval and MyRectangle classes to "factor out" the common features into the abstract class MyBoundedShape to produce the hierarchy in Fig. 10.18.

Class MyBoundedShape should declare two constructors that mimic those of class MyShape, only with an added parameter to set whether the shape is filled. Class MyBoundedShape should also declare *get* and *set* methods for manipulating the filled flag and methods that calculate the upper-left *x*-coordinate, upper-left *y*-coordinate, width and height. Remember, the values needed to draw an oval or a rectangle can be calculated from two *(x, y)* coordinates. If designed properly, the new `MyOval` and `MyRectangle` classes should each have two constructors and a `draw` method.

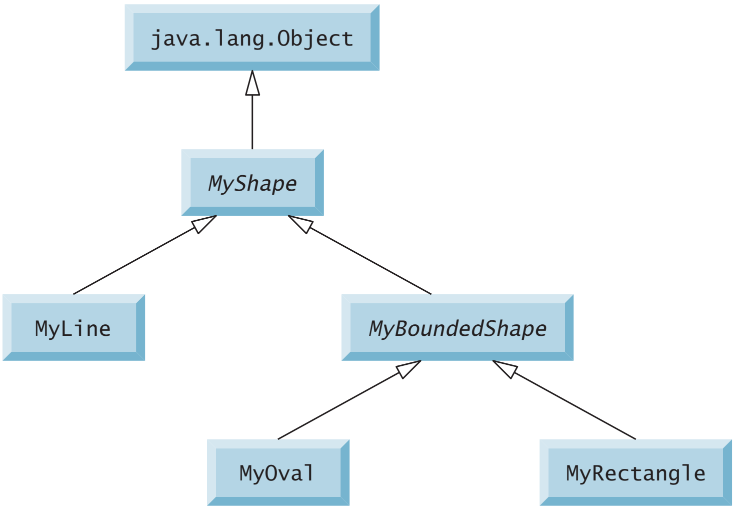


Fig. 10.18 | `MyShape` hierarchy with `MyBoundedShape`.

10.9 Wrap-Up

This chapter introduced polymorphism—the ability to process objects that share the same superclass in a class hierarchy as if they're all objects of the superclass. The chapter discussed how polymorphism makes systems extensible and maintainable, then demonstrated how to use overridden methods to effect polymorphic behavior. We introduced abstract classes, which allow you to provide an appropriate superclass from which other classes can inherit. You learned that an abstract class can declare abstract methods that each subclass must implement to become a concrete class and that a program can use variables of an abstract class to invoke the subclasses' implementations of abstract methods polymorphically. You also learned how to determine an object's type at execution time. We discussed the concepts of `final` methods and classes. Finally, the chapter discussed declaring and implementing an interface as another way to achieve polymorphic behavior.

You should now be familiar with classes, objects, encapsulation, inheritance, interfaces and polymorphism—the most essential aspects of object-oriented programming.

In the next chapter, you'll learn about exceptions, useful for handling errors during a program's execution. Exception handling provides for more robust programs.

Summary

Section 10.1 Introduction

- Polymorphism (p. 395) enables us to write programs that process objects that share the same superclass as if they're all objects of the superclass; this can simplify programming.
- With polymorphism, we can design and implement systems that are easily extensible. The only parts of a program that must be altered to accommodate new classes are those that require direct knowledge of the new classes that you add to the hierarchy.

Section 10.3 Demonstrating Polymorphic Behavior

- When the compiler encounters a method call made through a variable, the compiler determines if the method can be called by checking the variable's class type. If that class contains the proper method declaration (or inherits one), the call is compiled. At execution time, the type of the object to which the variable refers determines the actual method to use.

Section 10.4 Abstract Classes and Methods

- Abstract classes (p. 400) cannot be used to instantiate objects, because they're incomplete.
- The primary purpose of an abstract class is to provide an appropriate superclass from which other classes can inherit and thus share a common design.
- Classes that can be used to instantiate objects are called concrete classes (p. 401). They provide implementations of every method they declare (some of the implementations can be inherited).
- Programmers often write client code that uses only abstract superclasses (p. 401) to reduce client code's dependencies on specific subclass types.
- Abstract classes sometimes constitute several levels of a hierarchy.
- An abstract class normally contains one or more abstract methods (p. 401).
- Abstract methods do not provide implementations.
- A class that contains any abstract methods must be declared as an `abstract` class (p. 401). Each concrete subclass must provide implementations of each of the superclass's abstract methods.
- Constructors and `static` methods cannot be declared `abstract`.
- Abstract superclass variables can hold references to objects of any concrete class derived from the superclass. Programs typically use such variables to manipulate subclass objects polymorphically.
- Polymorphism is particularly effective for implementing layered software systems.

Section 10.5 Case Study: Payroll System Using Polymorphism

- A hierarchy designer can demand that each concrete subclass provide an appropriate method implementation by including an `abstract` method in a superclass.
- Most method calls are resolved at execution time, based on the type of the object being manipulated. This process is known as dynamic binding (p. 416) or late binding.
- A superclass variable can be used to invoke only methods declared in the superclass.
- Operator `instanceof` (p. 416) determines if an object has the *is-a* relationship with a specific type.
- Every object in Java knows its own class and can access it through `Object` method `getClass` (p. 417), which returns an object of type `Class` (package `java.lang`).
- The *is-a* relationship applies only between the subclass and its superclasses, not vice versa.

Section 10.6 `final` Methods and Classes

- A method that's declared `final` (p. 418) in a superclass cannot be overridden in a subclass.
- Methods declared `private` are implicitly `final`, because you can't override them in a subclass.
- Methods that are declared `static` are implicitly `final`.
- A `final` method's declaration can never change, so all subclasses use the same implementation, and calls to `final` methods are resolved at compile time—this is known as static binding (p. 419).
- Since the compiler knows that `final` methods cannot be overridden, it can optimize programs by removing calls to `final` methods and replacing them with the expanded code of their declarations at each method-call location—a technique known as inlining the code.
- A class that's declared `final` cannot be a superclass (p. 419).
- All methods in a `final` class are implicitly `final`.

Section 10.7 Case Study: Creating and Using Interfaces

- An interface (p. 419) specifies *what* operations are allowed but not *how* they're performed.
- A Java interface describes a set of methods that can be called on an object.
- An interface declaration begins with the keyword `interface` (p. 420).
- All interface members must be `public`, and interfaces may not specify any implementation details, such as concrete method declarations and instance variables.
- All methods declared in an interface are implicitly `public abstract` methods and all fields are implicitly `public`, `static` and `final`.
- To use an interface, a concrete class must specify that it `implements` (p. 420) the interface and must declare each interface method with the signature specified in the interface declaration. A class that does not implement all the interface's methods must be declared `abstract`.
- Implementing an interface is like signing a contract with the compiler that states, "I will declare all the methods specified by the interface or I will declare my class `abstract`."
- An interface is typically used when disparate (i.e., unrelated) classes need to share common methods and constants. This allows objects of unrelated classes to be processed polymorphically—objects of classes that implement the same interface can respond to the same method calls.
- You can create an interface that describes the desired functionality, then implement the interface in any classes that require that functionality.
- An interface is often used in place of an `abstract` class when there's no default implementation to inherit—that is, no instance variables and no default method implementations.
- Like `public abstract` classes, interfaces are typically `public` types, so they're normally declared in files by themselves with the same name as the interface and the `.java` file-name extension.
- Java does not allow subclasses to inherit from more than one superclass, but it does allow a class to inherit from a superclass and implement more than one interface.
- All objects of a class that implement multiple interfaces have the *is-a* relationship with each implemented interface type.
- An interface can declare constants. The constants are implicitly `public`, `static` and `final`.

Self-Review Exercises

10.1 Fill in the blanks in each of the following statements:

a) If a class contains at least one abstract method, it's a(n) ________ class.

b) Classes from which objects can be instantiated are called ________ classes.

c) ________ involves using a superclass variable to invoke methods on superclass and subclass objects, enabling you to "program in the general."

d) Methods that are not interface methods and that do not provide implementations must be declared using keyword ________.

e) Casting a reference stored in a superclass variable to a subclass type is called ________.

10.2 State whether each of the statements that follows is *true* or *false*. If *false*, explain why.

a) All methods in an `abstract` class must be declared as `abstract` methods.

b) Invoking a subclass-only method through a subclass variable is not allowed.

c) If a superclass declares an `abstract` method, a subclass must implement that method.

d) An object of a class that implements an interface may be thought of as an object of that interface type.

Answers to Self-Review Exercises

10.1 a) abstract. b) concrete. c) Polymorphism. d) `abstract`. e) downcasting.

10.2 a) False. An abstract class can include methods with implementations and `abstract` methods. b) False. Trying to invoke a subclass-only method with a superclass variable is not allowed. c) False. Only a concrete subclass must implement the method. d) True.

Exercises

10.3 How does polymorphism enable you to program "in the general" rather than "in the specific"? Discuss the key advantages of programming "in the general."

10.4 What are abstract methods? Describe the circumstances in which an abstract method would be appropriate.

10.5 How does polymorphism promote extensibility?

10.6 Discuss four ways in which you can assign superclass and subclass references to variables of superclass and subclass types.

10.7 Compare and contrast abstract classes and interfaces. Why would you use an abstract class? Why would you use an interface?

10.8 *(Payroll System Modification)* Modify the payroll system of Figs. 10.4–10.9 to include `private` instance variable `birthDate` in class `Employee`. Use class `Date` of Fig. 8.7 to represent an employee's birthday. Add *get* methods to class `Date`. Assume that payroll is processed once per month. Create an array of `Employee` variables to store references to the various employee objects. In a loop, calculate the payroll for each `Employee` (polymorphically), and add a $100.00 bonus to the person's payroll amount if the current month is the one in which the `Employee`'s birthday occurs.

10.9 *(Project: Shape Hierarchy)* Implement the `Shape` hierarchy shown in Fig. 9.3. Each `TwoDimensionalShape` should contain method `getArea` to calculate the area of the two-dimensional shape. Each `ThreeDimensionalShape` should have methods `getArea` and `getVolume` to calculate the surface area and volume, respectively, of the three-dimensional shape. Create a program that uses an array of `Shape` references to objects of each concrete class in the hierarchy. The program should print a text description of the object to which each array element refers. Also, in the loop that processes all the shapes in the array, determine whether each shape is a `TwoDimensionalShape` or a `ThreeDimensionalShape`. If it's a `TwoDimensionalShape`, display its area. If it's a `ThreeDimensionalShape`, display its area and volume.

10.10 *(Payroll System Modification)* Modify the payroll system of Figs. 10.4–10.9 to include an additional `Employee` subclass `PieceWorker` that represents an employee whose pay is based on the number of pieces of merchandise produced. Class `PieceWorker` should contain `private` instance variables `wage` (to store the employee's wage per piece) and `pieces` (to store the number of pieces produced). Provide a concrete implementation of method `earnings` in class `PieceWorker` that calculates the employee's earnings by multiplying the number of pieces produced by the wage per piece. Create an array of `Employee` variables to store references to objects of each concrete class in the new `Employee` hierarchy. For each `Employee`, display its `String` representation and earnings.

10.11 *(Accounts Payable System Modification)* In this exercise, we modify the accounts payable application of Figs. 10.11–10.15 to include the complete functionality of the payroll application of Figs. 10.4–10.9. The application should still process two `Invoice` objects, but now should process one object of each of the four `Employee` subclasses. If the object currently being processed is a `BasePlusCommissionEmployee`, the application should increase the `BasePlusCommissionEmployee`'s base salary by 10%. Finally, the application should output the payment amount for each object. Complete the following steps to create the new application:

a) Modify classes `HourlyEmployee` (Fig. 10.6) and `CommissionEmployee` (Fig. 10.7) to place them in the `Payable` hierarchy as subclasses of the version of `Employee` (Fig. 10.13) that implements `Payable`. [*Hint:* Change the name of method `earnings` to `getPaymentAmount` in each subclass so that the class satisfies its inherited contract with interface `Payable`.]

b) Modify class `BasePlusCommissionEmployee` (Fig. 10.8) such that it extends the version of class `CommissionEmployee` created in part (a).

c) Modify `PayableInterfaceTest` (Fig. 10.15) to polymorphically process two `Invoice`s, one `SalariedEmployee`, one `HourlyEmployee`, one `CommissionEmployee` and one `BasePlusCommissionEmployee`. First output a `String` representation of each `Payable` object. Next, if an object is a `BasePlusCommissionEmployee`, increase its base salary by 10%. Finally, output the payment amount for each `Payable` object.

10.12 *(Accounts Payable System Modification)* It's possible to include the functionality of the payroll application (Figs. 10.4–10.9) in the accounts payable application without modifying `Employee` subclasses `SalariedEmployee`, `HourlyEmployee`, `CommissionEmployee` or `BasePlusCommissionEmplyee`. To do so, you can modify class `Employee` (Fig. 10.4) to implement interface `Payable` and declare method `getPaymentAmount` to invoke method `earnings`. Method `getPaymentAmount` would then be inherited by the subclasses in the `Employee` hierarchy. When `getPaymentAmount` is called for a particular subclass object, it polymorphically invokes the appropriate `earnings` method for that subclass. Reimplement Exercise 10.11 using the original `Employee` hierarchy from the payroll application of Figs. 10.4–10.9. Modify class `Employee` as described in this exercise, and *do not* modify any of class `Employee`'s subclasses.

Making a Difference

10.13 *(`CarbonFootprint` Interface: Polymorphism)* Using interfaces, as you learned in this chapter, you can specify similar behaviors for possibly disparate classes. Governments and companies worldwide are becoming increasingly concerned with carbon footprints (annual releases of carbon dioxide into the atmosphere) from buildings burning various types of fuels for heat, vehicles burning fuels for power, and the like. Many scientists blame these greenhouse gases for the phenomenon called global warming. Create three small classes unrelated by inheritance—classes `Building`, `Car` and `Bicycle`. Give each class some unique appropriate attributes and behaviors that it does not have in common with other classes. Write an interface `CarbonFootprint` with a `getCarbonFootprint` method. Have each of your classes implement that interface, so that its `getCarbonFootprint` method calculates an appropriate carbon footprint for that class (check out a few websites that explain how to calculate carbon footprints). Write an application that creates objects of each of the three classes, places references to those objects in `ArrayList<CarbonFootprint>`, then iterates through the `ArrayList`, polymorphically invoking each object's `getCarbonFootprint` method. For each object, print some identifying information and the object's carbon footprint.