10
Object-Oriented Programming: Polymorphism

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.

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

10.1 Introduction

We now continue our study of object-oriented programming by explaining and demonstrating polymorphism with inheritance hierarchies. Polymorphism enables us to "program in the general" rather than "program in the specific." In particular, polymorphism enables us to write programs that process objects that share the same superclass in a class hierarchy as if they are 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 three types of animals under investigation. Imagine that each of these classes 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 array of references to objects of the various Animal subclasses.
Chapter 10 Object-Oriented Programming: Polymorphism

To simulate the animals’ movements, the program sends each object the same message once per second—namely, move. However, each specific type of Animal responds to a move message in a unique way—a Fish might swim three feet, a Frog might jump five feet and a Bird might fly ten feet. The program issues the same message (i.e., move) to each animal object generically, but 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.

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 to accommodate new classes are those that require direct knowledge of the new classes that the programmer adds 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 process each Animal generically can remain the same.

This chapter has several parts. First, we discuss common examples of polymorphism. We then provide an example demonstrating polymorphic behavior. We’ll 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 so-called time-and-a-half for overtime). We declare a common set of functionality for all the classes in the updated hierarchy in a so-called abstract class, Employee, from which classes SalariedEmployee, HourlyEmployee and CommissionEmployee inherit directly and 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 calculation is performed due to Java’s polymorphic capabilities.

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 use these capabilities to determine whether a particular employee object is a BasePlusCommissionEmployee. If so, we increase that employee’s base salary by 10%.

Next, the chapter introduces interfaces. An interface describes methods that can be called on an object, but does not provide concrete method implementations. 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 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 the same 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

Let’s consider several other examples of polymorphism. If class Rectangle is derived from class Quadrilateral, then a Rectangle object is a more specific version of a Quadrilateral object. Any operation (e.g., calculating the perimeter or the area) that can be performed on a Quadrilateral object can also be performed on a Rectangle object. 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 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.

As another example, suppose we design a video game that manipulates objects of classes Martian, Venusian, Plutonian, SpaceShip and LaserBeam. Imagine that each class inherits from the common superclass called SpaceObject, which contains method draw. Each subclass implements this method. A screen-manager program 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 in a unique way. 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 must build a class Mercurian that extends SpaceObject and provides its own draw method implementation. When objects of class Mercurian 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. Thus, without modifying the system (other than to build new classes and modify the code that creates new objects), programmers can use polymorphism to conveniently include additional types that were not envisioned when the system was created.

With polymorphism, the same method name and signature can be used to cause different actions to occur, depending on the type of object on which the method is invoked. This gives the programmer tremendous expressive capability.

Software Engineering Observation 10.1

Polymorphism enables programmers to deal in generalities and let the execution-time environment handle the specifics. Programmers can command objects to behave in manners
appropriate to those objects, without knowing the types of the objects (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 requiring modification of 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 commission employee class hierarchy, in which class `BasePlusCommissionEmployee` inherited from class `CommissionEmployee`. The examples in that section manipulated `CommissionEmployee` and `BasePlusCommissionEmployee` objects by using references to them to invoke their methods—we aimed superclass references at superclass objects and subclass references at subclass objects. These assignments are natural and straightforward—superclass references are intended to refer to superclass objects, and subclass references 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 _actual referenced object_, not the type of the _reference_, determines which method is called. This example demonstrates the key concept that an object of a subclass can be treated as an object of its superclass. This enables various interesting manipulations. A program can create an array of superclass references 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, we 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 from a subclass to its direct (and indirect) superclasses, and not vice versa.

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 detail 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 concrete example of downcasting 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 we assign 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. [Note: This program uses classes CommissionEmployee3 and BasePlusCommissionEmployee4 from Fig. 9.12 and Fig. 9.13, respectively.]

In Fig. 10.1, lines 10–11 create a CommissionEmployee3 object and assign its reference to a CommissionEmployee3 variable. Lines 14–16 create a BasePlusCommissionEmployee4 object and assign its reference to a BasePlusCommissionEmployee4 variable. These assignments are natural—for example, a CommissionEmployee3 variable's primary purpose is to hold a reference to a CommissionEmployee3 object. Lines 19–21 use reference

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

class PolymorphismTest {
   public static void main( String args[] ) {
      // assign superclass reference to superclass variable
      CommissionEmployee3 commissionEmployee = new CommissionEmployee3(
         "Sue", "Jones", "222-22-2222", 10000, .06);

      // assign subclass reference to subclass variable
      BasePlusCommissionEmployee4 basePlusCommissionEmployee =
         new BasePlusCommissionEmployee4(
         "Bob", "Lewis", "333-33-3333", 5000, .04, 300);

      // invoke toString on superclass object using superclass variable
      System.out.printf( "%s%n%n%n%n",
         "Call CommissionEmployee3's toString with superclass reference",
         "to superclass object", commissionEmployee.toString());

      // invoke toString on subclass object using subclass variable
      System.out.printf( "%s%n%n%n%n",
         "Call BasePlusCommissionEmployee4's toString with subclass",
         "reference to subclass object",
         basePlusCommissionEmployee.toString());

      // invoke toString on subclass object using superclass variable
      CommissionEmployee3 commissionEmployee2 =
         basePlusCommissionEmployee;
      System.out.printf( "%s%n%n%n%n",
         "Call BasePlusCommissionEmployee4's toString with superclass",
         "reference to subclass object", commissionEmployee2.toString());
   }
}
```

Fig. 10.1 | Assigning superclass and subclass references to superclass and subclass variables. (Part 1 of 2.)
Call CommissionEmployee3’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

Call BasePlusCommissionEmployee4’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 BasePlusCommissionEmployee4’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.)

commissionEmployee to invoke toString explicitly. Because commissionEmployee refers to a CommissionEmployee3 object, superclass CommissionEmployee3’s version of toString is called. Similarly, lines 24–27 use basePlusCommissionEmployee to invoke toString explicitly on the BasePlusCommissionEmployee4 object. This invokes subclass BasePlusCommissionEmployee4’s version of toString.

Lines 30–31 then assign the reference to subclass object basePlusCommissionEmployee to a superclass CommissionEmployee3 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 BasePlusCommissionEmployee4’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.

10.4 Abstract Classes and Methods

When we think of a class type, we assume that programs will create objects of that type. In some cases, however, it is useful to declare classes for which the programmer never in-
Abstract classes and methods tend to instantiate objects. Such classes are called abstract classes. Because they are 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." We demonstrate abstract classes in Section 10.5.

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—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, it 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 inheritance hierarchies contain abstract classes. However, programmers often write client code that uses only abstract superclass types to reduce client code's dependencies on a range of specific subclass types. For example, a programmer can write a method with a parameter of an abstract superclass type. When called, such a method can be passed 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 the hierarchy. For example, the Shape hierarchy of Fig. 9.3 begins with abstract class Shape. On the next level of the hierarchy are two more 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).

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 declared as an abstract class 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 static methods are inherited, they are not associated with particular objects of the classes that declare the static methods. Since abstract methods are meant to be overridden so 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 of the various classes in a class hierarchy. An abstract class typically contains one or more abstract methods that subclasses must override if the subclasses 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.

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. We 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 new shape types that the programmer will add to the system after writing the drawing program. The drawing program might need to display shapes, such as Circles, Triangles, Rectangles or others, that derive from abstract superclass 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 drawing program has ever encountered objects of that type.

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 subclasses that correspond to the specific types of device drivers. New devices are always being developed, and 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.
It is common in object-oriented programming to declare an *iterator class* that can traverse all the objects in a collection, such as an array (Chapter 7) or an ArrayList (Chapter 16, Collections). For example, a program can print an ArrayList of objects by creating an iterator object and using it to obtain the next list element each time the iterator is called. Iterators often are used in polymorphic programming to traverse a collection that contains references to objects from various levels of a hierarchy. (Chapter 16 presents a thorough treatment of ArrayList, iterators and “generics” capabilities.) An ArrayList of objects of class TwoDimensionalShape, for example, could contain objects from subclasses Square, Circle, Triangle and so on. Calling method draw for each TwoDimensionalShape object off a TwoDimensionalShape variable would polymorphically draw each object correctly on the screen.

### 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 the type of employee. We create an enhanced employee hierarchy to solve the following problem:

*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 for all hours worked in excess of 40 hours, commission employees are paid a percentage of their sales and 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 implement a Java 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. Note that abstract class Employee is italicized, as per the convention of the UML.
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” to something 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`.

**Software Engineering Observation 10.4**

A subclass can inherit “interface” or “implementation” from a superclass. Hierarchies designed for **implementation inheritance** tend to have their functionality high in the hierarchy—each new subclass inherits one or more methods that were implemented in a superclass, and the subclass uses the superclass implementations. Hierarchies designed for **interface inheritance** tend to have their functionality lower in the hierarchy—a superclass specifies one or more abstract methods that must be declared for each concrete class in the hierarchy, and the individual subclasses override these methods to provide subclass-specific implementations.

The following sections implement the `Employee` class hierarchy. Each of the first four sections implements 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 Creating 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 is not 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 a reference to the employee’s object to a superclass `Employee` variable, then invokes the `earnings` method on that variable. We maintain an array of `Employee` variables, each of which holds 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 iterates through the array and calls method `earnings` for each `Employee` object. Java processes these method calls polymorphically. Including `earnings` as an abstract method in `Employee` forces every direct subclass of `Employee` to override `earnings` in order to become a concrete class. This enables the designer of the class hierarchy to demand that each concrete subclass provide an appropriate pay calculation.

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...
10.5 Case Study: Payroll System Using Polymorphism

shows the desired results of each method. [Note: We do not list superclass Employee’s get and set methods because they are not overridden in any of the subclasses—each of these methods is inherited and used “as is” by each of the subclasses.]

Let us 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 25–26, 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 55–59), which returns the string representation of Employee; and abstract method earnings (line 62), which will be implemented by subclasses. Note that the Employee constructor does not validate the social security number in this example. Normally, such validation should be provided.

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 Employee type 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.
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

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

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

// set last name
public void setLastName(String last)
{
    lastName = last;
} // 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

Fig. 10.4 | Employee abstract superclass. (Part 1 of 2.)
### 10.5.2 Creating Concrete Subclass SalariedEmployee

Class SalariedEmployee (Fig. 10.5) extends class Employee (line 4) and overrides earnings (lines 29–32), 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 17–20); a get method to return weeklySalary's value (lines 23–26); a method earnings (lines 29–32) to calculate a SalariedEmployee's earnings; and a method toString (lines 35–39), 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 abstract method earnings in Employee 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, a compilation error occurs (and, of course, we want SalariedEmployee here to be a concrete class).
Method toString (lines 35–39) of class SalariedEmployee 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 38)—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 Creating 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 19–22 and 31–35 declare set methods that assign new values to instance variables wage and hours, respectively. Method setWage (lines 19–22) ensures that wage is nonnegative, and method setHours (lines 31–35) ensures that hours is between 0 and 168 (the total number of hours in a week) inclusive. Class HourlyEmployee also includes get methods (lines 25–28 and 38–41) to return the values of wage and hours, respectively; a method earnings (lines 44–50) to calculate an HourlyEmployee’s earnings; and a method toString (lines 53–58), which returns
the employee’s type, namely, “hourly employee:” and Employee-specific information. Note that 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 56) 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
// Fig. 10.6: HourlyEmployee.java
// HourlyEmployee class extends Employee.

class HourlyEmployee extends Employee
{
  private double wage; // wage per hour
  private double hours; // hours worked for week

  // five-argument constructor
  public HourlyEmployee( String first, String last, String ssn,
      double hourlyWage, double hoursWorked )
  {
    super( first, last, ssn );
    setWage( hourlyWage ); // validate hourly wage
    setHours( hoursWorked ); // validate hours worked
  } // end five-argument HourlyEmployee constructor

  // set wage
  public void setWage( double hourlyWage )
  {
    wage = ( hourlyWage < 0.0 ) ? 0.0 : hourlyWage;
  } // end method setWage

  // return wage
  public double getWage()
  {
    return wage;
  } // end method getWage

  // set hours worked
  public void setHours( double hoursWorked )
  {
    hours = ( ( hoursWorked >= 0.0 ) && ( hoursWorked <= 168.0 ) ) ?
        hoursWorked : 0.0;
  } // end method setHours

  // return hours worked
  public double getHours()
  {
    return hours;
  } // end method getHours

  // calculate earnings; override abstract method earnings in Employee
  public double earnings()
  {
    // code to calculate earnings
  } // end method earnings

Fig. 10.6 | HourlyEmployee class derived from Employee. (Part 1 of 2.)
```
10.5.4 Creating 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 19–22 and 31–34) to assign new values to instance variables commissionRate and grossSales, respectively; get methods (lines 25–28 and 37–40) that retrieve the values of these instance variables; method earnings (lines 43–46) to calculate a CommissionEmployee's earnings; and method toString (lines 49–55), 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 52) to obtain the Employee-specific information (i.e., first name, last name and social security number).

```java
if (getHours() <= 40) // no overtime
    return getWage() * getHours();
else
    return 40 * getWage() + (getHours() - 40) * getWage() * 1.5;
} // end method earnings

public String toString()
{
    return String.format("hourly employee: %s
%s: $%,.2f; %s: %,.2f", 
        super.toString(), "hourly wage", getWage(), 
        "hours worked", getHours());
} // end method toString

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

Fig. 10.7 | CommissionEmployee class derived from Employee. (Part 1 of 2.)
```
10.5.5 Creating 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 the first name, last name, social security number, sales amount and commission rate to the CommissionEmployee constructor (line 12) to initialize the inherited members. BasePlusCommissionEmployee also contains a set method (lines 17–20) to assign a new value to instance variable baseSalary and a get method (lines 23–26) to return baseSalary's
value. Method *earnings* (lines 29–32) calculates a *BasePlusCommissionEmployee*’s *earnings*. Note that line 31 in method *earnings* calls superclass *CommissionEmployee*’s *earnings* method to calculate the commission-based portion of the employee’s earnings. This is a nice example of code reuse. *BasePlusCommissionEmployee*’s *toString* method (lines 35–40) 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. Note that *BasePlusCommissionEmployee*’s *toString* initiates a chain of method calls that span all three levels of the Employee hierarchy.

```java
// Fig. 10.8: BasePlusCommissionEmployee.java
// BasePlusCommissionEmployee class extends CommissionEmployee.
public class BasePlusCommissionEmployee extends CommissionEmployee {
    private double baseSalary; // base salary per week

    // six-argument constructor
    public BasePlusCommissionEmployee( String first, String last,
     String ssn, double sales, double rate, double salary )
    {
        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 )
    {
        baseSalary = ( salary < 0.0 ) ? 0.0 : salary; // non-negative
    } // end method setBaseSalary

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

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

    // return String representation of BasePlusCommissionEmployee object
    public String toString()
    {
    }
}
```

Fig. 10.8 | *BasePlusCommissionEmployee* derives from *CommissionEmployee*. (Part 1 of 2.)
10.5 Case Study: Payroll System Using Polymorphism

10.5.6 Demonstrating 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, first 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, of course, 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. Note that each object’s toString method is called implicitly by printf when the object is output as a String with the %s format specifier.

```java
public class PayrollSystemTest {
    public static void main( String args[] ) {
        // create subclass objects
        SalariedEmployee salariedEmployee = new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
        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;

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

// generically process each element in array employees
for ( Employee currentEmployee : employees )
{
  System.out.println(  ); // invokes toString
  // determine whether element is a BasePlusCommissionEmployee
  if ( currentEmployee instanceof BasePlusCommissionEmployee )
  {
    // downcast Employee reference to
    // BasePlusCommissionEmployee reference
    BasePlusCommissionEmployee employee = (BasePlusCommissionEmployee) currentEmployee;

    double oldBaseSalary = employee.getBaseSalary();
    employee.setBaseSalary( 1.10 * oldBaseSalary );
    System.out.printf("new base salary with 10\% increase is: $%,.2f\n", employee.getBaseSalary() );
  } // end if

  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.println("Employee %d is a %s\n", j, employees[ j ].getClass().getName() );

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

Fig. 10.9  | Employee class hierarchy test program. (Part 2 of 3.)
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

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

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

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]. Each assignment is 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.
Lines 44–65 iterate through array \textbf{employees} and invoke methods \texttt{toString} and \texttt{earnings} with \texttt{Employee} control variable \texttt{currentEmployee}. The output illustrates that the appropriate methods for each class are indeed invoked. All calls to method \texttt{toString} and \texttt{earnings} are resolved at execution time, based on the type of the object to which \texttt{currentEmployee} refers. This process is known as \textbf{dynamic binding or late binding}. For example, line 46 implicitly invokes method \texttt{toString} of the object to which \texttt{currentEmployee} refers. As a result of dynamic binding, Java decides which class's \texttt{toString} method to call at execution time rather than at compile time. Note that only the methods of class \texttt{Employee} can be called via an \texttt{Employee} variable (and \texttt{Employee}, of course, includes the methods of class \texttt{Object}). (Section 9.7 discusses the set of methods that all classes inherit from class \texttt{Object}.) A superclass reference can be used to invoke only methods of the superclass (and the superclass can invoke overridden versions of these in the subclass).

We perform special processing on \texttt{BasePlusCommissionEmployee} objects—as we encounter these objects, 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 \texttt{Employee} object at execution time. Line 49 uses the \texttt{instanceof} operator to determine whether a particular \texttt{Employee} object's type is \texttt{BasePlusCommissionEmployee}. The condition in line 49 is true if the object referenced by \texttt{currentEmployee} is a \texttt{BasePlusCommissionEmployee}. This would also be true for any object of a \texttt{BasePlusCommissionEmployee} subclass because of the \texttt{is-a} relationship a subclass has with its superclass. Lines 53–54 downcast \texttt{currentEmployee} from type \texttt{Employee} to type \texttt{BasePlusCommissionEmployee}—this cast is allowed only if the object has an \texttt{is-a} relationship with \texttt{BasePlusCommissionEmployee}. The condition at line 49 ensures that this is the case. This cast is required if we are to invoke subclass \texttt{BasePlusCommissionEmployee} methods \texttt{getBaseSalary} and \texttt{setBaseSalary} on the current \texttt{Employee} object—as you'll see momentarily, attempting to invoke a subclass-only method directly on a superclass reference is a compilation error.

\begin{itemize}
\item \textbf{Common Programming Error 10.3} \hfill \texttt{Assigning} a superclass variable to a subclass variable (without an explicit cast) is a \texttt{ClassCastException}. \hfill \texttt{error.}
\item \textbf{Software Engineering Observation 10.5} \hfill \texttt{If} at execution time the reference of a subclass object has been assigned to a variable of one of its direct or indirect superclasses, it is acceptable to cast the reference stored in that superclass variable back to a reference of the subclass type. Before performing such a cast, use the \texttt{instanceof} operator to ensure that the object is indeed an object of an appropriate subclass type. \hfill \texttt{Before performing such a cast, use the} \texttt{instanceof} \texttt{operator to ensure that the object is indeed an object of an appropriate subclass type.}
\item \textbf{Common Programming Error 10.4} \hfill \texttt{When} downcasting an object, a \texttt{ClassCastException} occurs if at execution time the object does not have an \texttt{is-a} relationship with the type specified in the cast operator. An object can be cast only to its own type or to the type of one of its superclasses. \hfill \texttt{When downcasting an object, a} \texttt{ClassCastException} \texttt{occurs if at execution time the object does not have an} \texttt{is-a} \texttt{relationship with the type specified in the cast operator. An object can be cast only to its own type or to the type of one of its superclasses.}
\end{itemize}

If the \texttt{instanceof} expression in line 49 is true, the body of the \texttt{if} statement (lines 49–61) performs the special processing required for the \texttt{BasePlusCommissionEmployee} object. Using \texttt{BasePlusCommissionEmployee} variable \texttt{employee}, lines 56 and 57 invoke subclass-only methods \texttt{getBaseSalary} and \texttt{setBaseSalary} to retrieve and update the employee's base salary with the 10% raise.
Lines 63–64 invoke method `earnings` on `currentEmployee`, which calls the appropriate subclass object’s `earnings` method polymorphically. As you can see, obtaining the earnings of the `SalariedEmployee`, `HourlyEmployee` and `CommissionEmployee` polymorphically in lines 63–64 produces the same result as obtaining these employees’ earnings individually in lines 22–27. However, 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.

Lines 68–70 display each employee’s type as a string. Every object in Java knows its own class and can access this information through the `getClass` method, which all classes inherit from class `Object`. The `getClass` method 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 the `getClass` method on the object to get its runtime class (i.e., a `Class` object that represents the object’s type). Then method `getName` is invoked on the object returned by `getClass` to get the class’s name. To learn more about class `Class`, see its online documentation at [java.sun.com/javase/6/docs/api/java/lang/Class.html](http://java.sun.com/javase/6/docs/api/java/lang/Class.html).

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 `commissionEmployee` to subclass variable `basePlusCommissionEmployee` 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, 57 and 60 used superclass variable `currentEmployee`, rather than subclass variable `employee`, to invoke subclass-only methods `getBaseSalary` and `setBaseSalary`, we would receive a “cannot find symbol” compilation error on each of these lines. Attempting to invoke subclass-only methods on a superclass reference is not allowed. While lines 56, 57 and 60 execute only if `instanceof` in line 49 returns true to indicate that `currentEmployee` has been assigned a reference to a `BasePlusCommissionEmployee` object, we cannot attempt to invoke subclass `BasePlusCommissionEmployee` methods `getBaseSalary` and `setBaseSalary` on superclass `Employee` reference `currentEmployee`. The compiler would generate errors in lines 56, 57 and 60, because `getBaseSalary` and `setBaseSalary` are not superclass methods and cannot be invoked on a superclass variable. Although the actual method that is called depends on the object’s type at execution time, a variable can be used to invoke only those methods that are members of that variable’s type, which the compiler verifies. Using a superclass `Employee` variable, we can invoke only methods found in class `Employee`—`earnings`, `toString` and `Employee`’s `get` and `set` methods.

### 10.5.7 Summary of the Allowed Assignments Between Superclass and Subclass Variables

Now that you have 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 if they are
superclass objects. 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.

In the current section and in Section 10.3 and Chapter 9, we have 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, this reference 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 13.) The instanceof operator can be used 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 Section 6.10 that variables can be declared final to indicate that they cannot be modified after they are initialized—such variables represent constant values. It is also possible to declare methods, method parameters and classes with the final modifier.

A method that is declared final in a superclass cannot be overridden in a subclass. Methods that are declared private are implicitly final, because it is impossible 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. 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.

Performance Tip 10.1

The compiler can decide to inline a final method call and will do so for small, simple final methods. Inlining does not violate encapsulation or information hiding, but does improve performance because it eliminates the overhead of making a method call.

A class that is 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. This class cannot be extended, so programs that use Strings 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 information on final classes and methods, visit java.sun.com/docs/books/tutorial/java/IandI/final.html. This site contains additional insights into using final classes to improve the security of a system.
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—the fundamental capabilities of object-oriented programming. However, in some cases, it is important to declare classes final—typically for security reasons.

10.7 Case Study: Creating and Using Interfaces

Our next example (Figs. 10.11–10.13) 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 that requires that unrelated classes implement a set of common methods (e.g., a method that calculates a payment amount)? Java interfaces offer exactly this capability.

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., changing the station, adjusting the volume, choosing 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. Similarly, the interface between a driver and a car with a manual transmission includes the steering wheel, the gear shift, the clutch pedal, the gas pedal and the brake pedal. This same interface is found in nearly all manual transmission cars, enabling someone who knows how to drive one particular manual transmission car to drive just about any manual transmission car. The components of each individual car may look different, but their general purpose is the same—to allow people to drive the car.

Software objects also communicate via interfaces. A Java interface describes a set of methods that can be called on an object, to tell the object to perform some task or return some piece of information, for example. 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. So all methods declared in an interface are implicitly public abstract methods and all fields are implicitly public, static and final.
Chapter 10  Object-Oriented Programming: Polymorphism

**Good Programming Practice 10.1**

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

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. 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`.

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 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`).

An interface is often used in place of an `abstract` class when there is no default implementation to inherit—that is, no fields and no default method implementations. Interfaces are typically `public` types, so they are normally declared in files by themselves with the same name as the interface and the `.java` file-name extension.

### 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 (i.e., we rename `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 `Payable`, 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 our company’s accounts payable application.

The UML class diagram in Fig. 10.10 shows the hierarchy used in our accounts payable application. The UML 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 a 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. Note that, as in the class diagram of Fig. 10.2, class Employee appears in italics, indicating that it is 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 Declaring Interface Payable

The declaration of interface Payable begins in Fig. 10.11 at line 4. Interface Payable contains public abstract method getPaymentAmount (line 6). Note that the method is not explicitly declared public or abstract. Interface methods must be 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. (We’ll see later in the book the notion of “tagging interfaces”—these actually have no methods. In fact, a tagging interface contains no constant values, either—it simply contains an empty interface declaration.) In addition, method getPaymentAmount has no parameters, but interface methods can have parameters.

```java
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 Creating 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 22–67) that manipulate the class’s instance variables and a toString method (lines 70–75) that returns a string representation of an Invoice object. Note that methods setQuantity (lines 46–49) and setPricePerItem (lines 58–61) ensure that quantity and pricePerItem obtain only nonnegative values.

```java
// Fig. 10.12: Invoice.java
// Invoice class 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;
    } // end method setPartNumber

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

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

Fig. 10.12  |  Invoice class that implements Payable. (Part 1 of 2.)
Line 4 of Fig. 10.12 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 does allow a class to inherit from a superclass and implement more than one interface. In fact, a class can implement as many interfaces as it needs, in addition to extending one other class. To implement more than one inter-
face, 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, ...

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 78–81. 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 80). This method satisfies the implementation requirement for this method in interface Payable—we have 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 Employee class. This class declaration is identical to that of Fig. 10.4 with only two exceptions. First, line 4 of Fig. 10.13 indicates that class Employee now implements interface Payable. Second, since Employee now implements interface Payable, 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—first we must know the specific type of Employee. In Fig. 10.4, we declared method earnings as abstract for this reason, and as a result class Employee had to be declared abstract. This forced each Employee subclass to override earnings with a concrete implementation.

```java
// Fig. 10.13: Employee.java
// Employee abstract superclass implements Payable.

public abstract class Employee implements Payable {
    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

    // set first name
    public void setFirstName( String first ) {
```

Fig. 10.13 | Employee class that implements Payable. (Part 1 of 2.)
In Fig. 10.13, we handle this situation differently. Recall that when a class implements an interface, the class 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 are 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.
abstract. As indicated by the comments in lines 61–62, 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 version of class SalariedEmployee that extends Employee and fulfills superclass Employee’s contract to implement method getPaymentAmount of interface Payable. This version of SalariedEmployee is identical to that of Fig. 10.5 with the exception that the version here implements method getPaymentAmount (lines 30–33) instead of method earnings. The two methods contain the same functionality but have different names. 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 and use only SalariedEmployee in our test program in this section.
When a class implements an interface, the same *is-a* relationship provided by inheritance applies. For example, 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. As with inheritance relationships, an object of a class that implements an interface may be thought of as an object of the interface type. 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 10.7**

Inheritance and interfaces are similar in their implementation of the *is-a* relationship. An object of a class that implements an interface may be thought of as an object of that interface type. An object of any subclasses of a class that implements an interface also can be thought of as an object of the interface type.

**Software Engineering Observation 10.8**

The *is-a* relationship that exists between superclasses and subclasses, and between interfaces and the classes that implement them, holds when passing an object to a method. When a method parameter receives a variable of 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 specified in the superclass declaration (and in class *Object*). Using an interface reference, we can polymorphically invoke any method specified in the interface declaration (and in class *Object*—because a variable of an interface type must refer to an object to call methods, and all objects contain the methods of class *Object*).
10.7.6 Using Interface Payable to Process Invoices and Employees Polymorphically

`PayableInterfaceTest` (Fig. 10.15) illustrates that interface Payable can be used to process a set of Invoices and Employees 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. Note that line 27 invokes method `toString` off 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. (Note that `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 `currentEmployee` refers to an Invoice during the first iteration of the for loop, class Invoice’s `toString` and `getPaymentAmount` execute.

**Software Engineering Observation 10.10**

All methods of class `Object` can be called by using a reference of an interface type. A reference refers to an object, and all objects inherit the methods of class `Object`.

```java
// Fig. 10.15: PayableInterfaceTest.java
// Tests interface Payable.
public class PayableInterfaceTest {
    public static void main( String args[] ) {
        // create four-element Payable array
        Payable payableObjects[] = new Payable[ 4 ];

        // populate array with objects that implement Payable
        payableObjects[ 0 ] = new Invoice( "01234", "seat", 2, 375.00 );
        payableObjects[ 1 ] = new Invoice( "56789", "tire", 4, 79.95 );
        payableObjects[ 2 ] =
            new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
        payableObjects[ 3 ] =
            new SalariedEmployee( "Lisa", "Barnes", "888-88-8888", 1200.00 );

        System.out.println( 
            "Invoices and Employees processed polymorphically:\n" );
    }
}
```

Fig. 10.15 | Payable interface test program processing Invoices and Employees polymorphically. (Part 1 of 2.)
10.7 Case Study: Creating and Using Interfaces

10.7.7 Declaring Constants with Interfaces

As we mentioned an interface can declare constants. The constants are implicitly `public`, `static` and `final`—again, these keywords are not required in the interface declaration. One popular use of an interface is to declare a set of constants that can be used in many class declarations. Consider interface `Constants`:

```
public interface Constants {
    int ONE = 1;
    int TWO = 2;
    int THREE = 3;
}
```

A class can use these constants by importing the interface, then referring to each constant as `Constants.ONE`, `Constants.TWO` and `Constants.THREE`. Note that a class can refer to the imported constants with just their names (i.e., `ONE`, `TWO` and `THREE`) if it uses a static import declaration (presented in Section 8.12) to import the interface.

```
// generically process each element in array payableObjects
for ( Payable currentPayable : payableObjects ) {
    // output currentPayable and its appropriate payment amount
    System.out.printf("%s
%s
\n\",
        currentPayable.toString(),
        "payment due", currentPayable.getPaymentAmount() );
} // end for
} // end main
} } // end class PayableInterfaceTest

Invoices and Employees processed polymorphically:

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

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.)
### Software Engineering Observation 10.11

As of Java SE 5.0, it became a better programming practice to create 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.

### 10.7.8 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). As you’ll see throughout this book, 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 presents a brief overview of a few of the more popular interfaces of the Java API.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparable</td>
<td>As you learned in Chapter 2, Java contains several comparison operators (e.g., <code>&lt;</code>, <code>&lt;=</code>, <code>&gt;</code>, <code>&gt;=</code>, <code>==</code>, <code>!=</code>) that allow you to compare primitive values. However, these operators cannot be used to compare the contents of objects. Interface <code>Comparable</code> is used to allow objects of a class that implements the interface to be compared to one another. The interface contains one method, <code>compareTo</code>, that compares the object that calls the method to the object passed as an argument to the method. Classes must implement <code>compareTo</code> such that it returns a value indicating whether the object on which it is invoked is less than (negative integer return value), equal to (0 return value) or greater than (positive integer return value) the object passed as an argument, using any criteria specified by the programmer. For example, if class <code>Employee</code> implements <code>Comparable</code>, its <code>compareTo</code> method could compare <code>Employee</code> objects by their earnings amounts. Interface <code>Comparable</code> is commonly used for ordering objects in a collection such as an array. We use <code>Comparable</code> in Chapter 15, Generics, and Chapter 16, Collections.</td>
</tr>
<tr>
<td>Serializable</td>
<td>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 <code>Serializable</code> in Chapter 14, Files and Streams, and Chapter 19, Networking.</td>
</tr>
<tr>
<td>Runnable</td>
<td>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 18, Multithreading). The interface contains one method, <code>run</code>, which describes the behavior of an object when executed.</td>
</tr>
<tr>
<td>GUI event-listener</td>
<td>You work with graphical user interfaces (GUIs) every day. For example, in your web browser, you might type in a text field the address of a website to visit, or you might click a button to return to the previous site you</td>
</tr>
</tbody>
</table>

---

Incorporating Inheritance into the ATM System

We now revisit our ATM system design to see how it might benefit from inheritance. To apply inheritance, we first look for commonality among classes in the system. We create an inheritance hierarchy to model similar (yet not identical) classes in a more elegant and efficient manner. We then modify our class diagram to incorporate the new inheritance relationships. Finally, we demonstrate how our updated design is translated into Java code.

In Section 3.9, we encountered the problem of representing a financial transaction in the system. Rather than create one class to represent all transaction types, we decided to create three individual transaction classes—BalanceInquiry, Withdrawal and Deposit—to represent the transactions that the ATM system can perform. Figure 10.17 shows the attributes and operations of classes BalanceInquiry, Withdrawal and Deposit. Note that these classes have one attribute (accountNumber) and one operation (execute) in common. Each class requires attribute accountNumber to specify the account to which the

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SwingConstants</td>
<td>Contains constants used in GUI programming to position GUI elements on the screen. We explore GUI programming in Chapters 11 and 17.</td>
</tr>
</tbody>
</table>

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

**10.8 (Optional) Software Engineering Case Study: Incorporating Inheritance into the ATM System**

We now revisit our ATM system design to see how it might benefit from inheritance. To apply inheritance, we first look for commonality among classes in the system. We create an inheritance hierarchy to model similar (yet not identical) classes in a more elegant and efficient manner. We then modify our class diagram to incorporate the new inheritance relationships. Finally, we demonstrate how our updated design is translated into Java code.

In Section 3.9, we encountered the problem of representing a financial transaction in the system. Rather than create one class to represent all transaction types, we decided to create three individual transaction classes—BalanceInquiry, Withdrawal and Deposit—to represent the transactions that the ATM system can perform. Figure 10.17 shows the attributes and operations of classes BalanceInquiry, Withdrawal and Deposit. Note that these classes have one attribute (accountNumber) and one operation (execute) in common. Each class requires attribute accountNumber to specify the account to which the

![Fig. 10.17](image-url) | Attributes and operations of classes BalanceInquiry, Withdrawal and Deposit.

transaction applies. Each class contains operation execute, which the ATM invokes to perform the transaction. Clearly, BalanceInquiry, Withdrawal and Deposit represent types of transactions. Figure 10.17 reveals commonality among the transaction classes, so using inheritance to factor out the common features seems appropriate for designing classes BalanceInquiry, Withdrawal and Deposit. We place the common functionality in a superclass, Transaction, that classes BalanceInquiry, Withdrawal and Deposit extend.

The UML specifies a relationship called a generalization to model inheritance. Figure 10.18 is the class diagram that models the generalization of superclass Transaction and subclasses BalanceInquiry, Withdrawal and Deposit. The arrows with triangular hollow arrowheads indicate that classes BalanceInquiry, Withdrawal and Deposit extend class Transaction. Class Transaction is said to be a generalization of classes BalanceInquiry, Withdrawal and Deposit. Class BalanceInquiry, Withdrawal and Deposit are said to be specializations of class Transaction.

Classes BalanceInquiry, Withdrawal and Deposit share integer attribute accountNumber, so we factor out this common attribute and place it in superclass Transaction. We no longer list accountNumber in the second compartment of each subclass, because the three subclasses inherit this attribute from Transaction. Recall, however, that subclasses cannot access private attributes of a superclass. We therefore include public method getAccountNumber in class Transaction. Each subclass will inherit this method, enabling the subclass to access its accountNumber as needed to execute a transaction.

According to Fig. 10.17, classes BalanceInquiry, Withdrawal and Deposit also share operation execute, so we decided that superclass Transaction should contain public method execute. However, it does not make sense to implement execute in class Transaction, because the functionality that this method provides depends on the type of the actual transaction. We therefore declare method execute as abstract in superclass Transaction. Any class that contains at least one abstract method must also be declared abstract. This forces any subclass of Transaction that must be a concrete class (i.e., BalanceInquiry, Withdrawal and Deposit) to implement method execute. The UML requires that we place abstract class names (and abstract methods) in italics, so Transaction and its method execute appear in italics in Fig. 10.18. Note that method execute is not italicized in subclasses BalanceInquiry, Withdrawal and Deposit. Each subclass overrides superclass Transaction's execute method with a concrete implementation that performs the steps appropriate for completing that type of transaction. Note that Fig. 10.18 includes operation execute in the third compartment of classes BalanceInquiry, Withdrawal and Deposit, because each class has a different concrete implementation of the overridden method.

Incorporating inheritance provides the ATM with an elegant way to execute all transactions "in the general." For example, suppose a user chooses to perform a balance inquiry. The ATM sets a Transaction reference to a new object of class BalanceInquiry. When the ATM uses its Transaction reference to invoke method execute, BalanceInquiry's version of execute is called.

This polymorphic approach also makes the system easily extensible. Should we wish to create a new transaction type (e.g., funds transfer or bill payment), we would just create an additional Transaction subclass that overrides the execute method with a version of the method appropriate for executing the new transaction type. We would need to make only minimal changes to the system code to allow users to choose the new transaction type
Incorporating Inheritance into the ATM System

from the main menu and for the ATM to instantiate and execute objects of the new subclass. The ATM could execute transactions of the new type using the current code, because it executes all transactions polymorphically using a general Transaction reference.

An abstract class like Transaction is one for which the programmer never intends to instantiate objects. An abstract class simply declares common attributes and behaviors of its subclasses in an inheritance hierarchy. Class Transaction defines the concept of what it means to be a transaction that has an account number and executes. You may wonder why we bother to include abstract method execute in class Transaction if it lacks a concrete implementation. Conceptually, we include this method because it corresponds to the defining behavior of all transactions—executing. Technically, we must include method execute in superclass Transaction so that the ATM (or any other class) can polymorphically invoke each subclass’s overridden version of this method through a Transaction reference. Also, from a software engineering perspective, including an abstract method in a superclass forces the implementor of the subclasses to override that method with concrete implementations in the subclasses, or else the subclasses, too, will be abstract, preventing objects of those subclasses from being instantiated.

Subclasses BalanceInquiry, Withdrawal and Deposit inherit attribute accountNumber from superclass Transaction, but classes Withdrawal and Deposit contain the additional attribute amount that distinguishes them from class BalanceInquiry. Classes Withdrawal and Deposit require this additional attribute to store the amount of money that the user wishes to withdraw or deposit. Class BalanceInquiry has no need for such an attribute and requires only an account number to execute. Even though two of the three Transaction subclasses share this attribute, we do not place it in superclass Transaction—we place only features common to all the subclasses in the superclass, otherwise subclasses could inherit attributes (and methods) that they do not need and should not have.

Figure 10.19 presents an updated class diagram of our model that incorporates inheritance and introduces class Transaction. We model an association between class ATM and class Transaction to show that the ATM, at any given moment is either executing a trans-
action or it is not (i.e., zero or one objects of type Transaction exist in the system at a time). Because a Withdrawal is a type of Transaction, we no longer draw an association line directly between class ATM and class Withdrawal. Subclass Withdrawal inherits superclass Transaction’s association with class ATM. Subclasses BalanceInquiry and Deposit inherit this association, too, so the previously omitted associations between ATM and classes BalanceInquiry and Deposit no longer exist either.

We also add an association between class Transaction and the BankDatabase (Fig. 10.19). All Transactions require a reference to the BankDatabase so they can access and modify account information. Because each Transaction subclass inherits this reference, we no longer model the association between class Withdrawal and the BankDatabase. Similarly, the previously omitted associations between the BankDatabase and classes BalanceInquiry and Deposit no longer exist.

We show an association between class Transaction and the Screen. All Transactions display output to the user via the Screen. Thus, we no longer include the association previously modeled between Withdrawal and the Screen, although Withdrawal still participates in associations with the CashDispenser and the Keypad. Our class diagram incorporating inheritance also models Deposit and BalanceInquiry. We show associations between Deposit and both the DepositSlot and the Keypad. Note that class BalanceInquiry.
Incorporating Inheritance into the ATM System

A BalanceInquiry takes part in no associations other than those inherited from class Transaction—a BalanceInquiry needs to interact only with the BankDatabase and with the Screen.

The class diagram of Fig. 8.21 showed attributes and operations with visibility markers. Now we present a modified class diagram that incorporates inheritance in Fig. 10.20. This abbreviated diagram does not show inheritance relationships, but instead shows the attributes and methods after we have employed inheritance in our system. To save space, as we did in Fig. 10.19—we do, however, include them in the Java implementation in Appendix H. We also omit all operation parameters, as we did in Fig. 8.21—incorporating inheritance does not affect the parameters already modeled in Figs. 6.27–6.30.

---

**Fig. 10.20**  
Class diagram with attributes and operations (incorporating inheritance). Note that abstract class names (e.g., Transaction) and method names (e.g., execute in class Transaction) appear in italics.

10.8 Incorporating Inheritance into the ATM System
Software Engineering Observation 10.12

A complete class diagram shows all the associations among classes and all the attributes and operations for each class. When the number of class attributes, methods and associations is substantial, a good practice that promotes readability is to divide this information between two class diagrams—one focusing on associations and the other on attributes and methods.

Implementing the ATM System Design (Incorporating Inheritance)

In Section 8.18, we began implementing the ATM system design. We now modify our implementation to incorporate inheritance, using class Withdrawal as an example.

1. If a class A is a generalization of class B, then class B extends class A in the class declaration. For example, abstract superclass Transaction is a generalization of class Withdrawal. Figure 10.21 contains the shell of class Withdrawal containing the appropriate class declaration.

2. If class A is an abstract class and class B is a subclass of class A, then class B must implement the abstract methods of class A if class B is to be a concrete class. For example, class Transaction contains abstract method execute, so class Withdrawal must implement this method if we want to instantiate a Withdrawal object. Figure 10.22 is the Java code for class Withdrawal from Fig. 10.19 and Fig. 10.20. Class Withdrawal inherits field accountNumber from superclass Transaction, so Withdrawal does not need to declare this field. Class Withdrawal also inherits references to the Screen and the BankDatabase from its superclass Transaction, so we do not include these references in our code. Figure 10.20 specifies attribute amount and operation execute for class Withdrawal. Line 6 of Fig. 10.22 declares a field for attribute amount. Lines 16–18 declare the shell of a method for operation execute. Recall that subclass Withdrawal must provide a concrete implementation of the abstract method execute in superclass Transaction. The keypad and cashDispenser references (lines 7–8) are fields derived from Withdrawal’s associations in Fig. 10.19. [Note: The constructor in the complete working version of this class will initialize these references to actual objects.]

```java
// Class Withdrawal represents an ATM withdrawal transaction
publlic class Withdrawal extends Transaction
{
}
// end class Withdrawal

Fig. 10.21 | Java code for shell of class Withdrawal.

// Withdrawal.java
// Generated using the class diagrams in Fig. 10.21 and Fig. 10.22
public class Withdrawal extends Transaction
{
    // attributes
    private double amount; // amount to withdraw
    private Keypad keypad; // reference to keypad
    private CashDispenser cashDispenser; // reference to cash dispenser

Fig. 10.22 | Java code for class Withdrawal based on Figs. 10.19 and 10.20. (Part 1 of 2.)
```
Several UML modeling tools convert UML-based designs into Java code and can speed the implementation process considerably. For more information on these tools, refer to the web resources listed at the end of Section 2.8.

Congratulations on completing the design portion of the case study! We completely implement the ATM system in 670 lines of Java code in Appendix H. We recommend that you carefully read the code and its description. The code is abundantly commented and precisely follows the design with which you are now familiar. The accompanying description is carefully written to guide your understanding of the implementation based on the UML design. Mastering this code is a wonderful culminating accomplishment after studying Chapters 1–8.

Software Engineering Case Study Self-Review Exercises

10.1 The UML uses an arrow with a _________ to indicate a generalization relationship.
   a) solid filled arrowhead
   b) triangular hollow arrowhead
   c) diamond-shaped hollow arrowhead
   d) stick arrowhead

10.2 State whether the following statement is true or false, and if false, explain why: The UML requires that we underline abstract class names and method names.

10.3 Write Java code to begin implementing the design for class Transaction specified in Figs. 10.19 and 10.20. Be sure to include private reference-type attributes based on class Transaction's associations. Also be sure to include public get methods that provide access to any of these private attributes that the subclasses require to perform their tasks.

Answers to Software Engineering Case Study Self-Review Exercises

10.1 b.

10.2 False. The UML requires that we italicize abstract class names and method names.

10.3 The design for class Transaction yields the code in Fig. 10.23. The bodies of the class constructor and methods will be completed in Appendix H. When fully implemented, methods getScreen and getBankDatabase will return superclass Transaction's private reference attributes screen and bankDatabase, respectively. These methods allow the Transaction subclasses to access the ATM's screen and interact with the bank's database.

Fig. 10.22 | Java code for class Withdrawal based on Figs. 10.19 and 10.20. (Part 2 of 2.)
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 are 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 programmers 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. 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, we take a deeper look at graphical user interfaces (GUIs).