10.3 Demonstrating Polymorphic Behavior

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

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

reference of a `BasePlusCommissionEmployee` object to a superclass `CommissionEmployee` variable, because a `BasePlusCommissionEmployee` *is a* `CommissionEmployee`—so we can treat a `BasePlusCommissionEmployee` as a `CommissionEmployee`.

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

The Java compiler *does* allow the assignment of a superclass reference to a subclass variable if we explicitly *cast* the superclass reference to the subclass type. 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 demonstrate the mechanics of downcasting in Section 10.5.

Software Engineering Observation 10.3

Although it's allowed, you should generally avoid downcasting.

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

```
1  // Fig. 10.1: PolymorphismTest.java
2  // Assigning superclass and subclass references t
3  // subclass variables.
```

```
                          4
        5  public class PolymorphismTest {
  6     public static void main(String[] args) {
  7       // assign superclass reference to superclas
  8       CommissionEmployee commissionEmployee = new
  9          "Sue", "Jones", "222-22-2222", 10000, .0
                          10
 11       // assign subclass reference to subclass va
 12       BasePlusCommissionEmployee basePlusCommissi
    13          new BasePlusCommissionEmployee(
 14          "Bob", "Lewis", "333-33-3333", 5000, .04
                          15
 16       // invoke toString on superclass object usi
  17       System.out.printf("%s %s:%n%n%s%n%n",
 18          "Call CommissionEmployee's toString with
 19          "to superclass object", commissionEmploy
                          20
 21       // invoke toString on subclass object using
  22       System.out.printf("%s %s:%n%n%s%n%n",
 23          "Call BasePlusCommissionEmployee's toStr
   24          "reference to subclass object",
 25          basePlusCommissionEmployee.toString());
                          26
 27       // invoke toString on subclass object using
 28       CommissionEmployee commissionEmployee2 =
     29          basePlusCommissionEmployee;
  30       System.out.printf("%s %s:%n%n%s%n",
 31          "Call BasePlusCommissionEmployee's toStr
 32          "reference to subclass object", commissi
                       33    }
                         34  }
```

```
Call CommissionEmployee's toString with superclass re

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

```
Call BasePlusCommissionEmployee's toString with subcl

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

Call BasePlusCommissionEmployee's toString with super

   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.

In Fig. 10.1, lines 8–9 create a `CommissionEmployee` object and assign its reference to a `CommissionEmployee` variable. Lines 12–14 create a `BasePlusCommissionEmployee` object and assign its reference to a `BasePlusCommissionEmployee` variable. These assignments are natural—for example, a `CommissionEmployee` variable's primary purpose is to hold a reference to a `CommissionEmployee` object. Lines 17–19 use `commissionEmployee` to invoke `toString` *explicitly*. Because `commissionEmployee` refers to a

`CommissionEmployee` object, superclass `CommissionEmployee`'s version of `toString` is called. Similarly, lines 22–25 use `basePlusCommissionEmployee` to invoke `toString` *explicitly* on the `Base-PlusCommissionEmployee` object. This invokes subclass `BasePlusCommissionEmployee`'s version of `toString`.

Lines 28–29 then assign the reference of subclass object `basePlusCommissionEmployee` to a superclass `CommissionEmployee` variable, which lines 30–32 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 32 actually calls class `BasePlusCommissionEmployee`'s `toString` method. The Java compiler allows this "crossover" because an object of a subclass *is an* object of its superclass (but *not* vice versa). When the compiler encounters a method call made through a variable, the compiler determines if the method can be called by checking the variable's class type. If that class contains the proper method declaration (or inherits one), the call is compiled. At execution time, the type of the object to which the variable refers determines the actual method to use. This process, called *dynamic binding*, is discussed in detail in Section 10.5.