EXAMPLE: Sales Records

Suppose you are designing a record-keeping program for an automobile parts store. You want to make the program versatile, but you are not sure you can account for all possible situations. For example, you want to keep track of sales, but you cannot anticipate all types of sales. At first, there will only be regular sales to retail customers who go to the store to buy one particular part. However, later you may want to add sales with discounts or mail order sales with a shipping charge. All of these sales will be for an item with a basic price and ultimately will produce some bill. For a simple sale, the bill is just the basic price, but if you later add discounts, then some kinds of bills will also depend on the size of the discount. Now your program needs to compute daily gross sales, which intuitively should just be the sum of all the individual sales bills. You may also want to calculate the largest and smallest sales of the day or the average sale for the day. All of these can be calculated from the individual bills, but many of the methods for computing the bills will not be added until later, when you decide what types of sales you will be dealing with. Because Java uses late binding, you can write a program to total all bills, even though you will not determine the code for some of the bills until later. (For simplicity in this first example, we assume that each sale is for just one item, although we could—but will not here—account for sales of multiple items.)

Display 8.1 contains the definition for a class named `Sale`. All types of sales will be derived classes of the class `Sale`. The class `Sale` corresponds to simple sales of a single item with no added discounts and no added charges. Note that the methods `lessThan` and `equalDeals` both include invocations of the method `bill`. We can later define derived classes of the class `Sale` and define their versions of the method `bill`, and the definitions of the methods `lessThan` and `equalDeals` (which we gave with the class `Sale`) will use the version of the method `bill` that corresponds to the object of the derived class.

For example, Display 8.2 shows the derived class `DiscountSale`. Notice that this class requires a different definition for its version of the method `bill`. Now the methods `lessThan` and `equalDeals`, which use the method `bill`, are inherited from the base class `Sale`. But, when the methods `lessThan` and `equalDeals` are used with an object of the class `DiscountSale`, they will use the version of the method definition for `bill` that was given with the class `DiscountSale`. This is indeed a pretty fancy trick for Java to pull off. Consider the method call `d1.lessThan(d2)` for objects `d1` and `d2` of the class `DiscountSale`. The definition of the method `lessThan` (even for an object of the class `DiscountSale`) is given in the definition of the base class `Sale`, which was compiled before we ever even thought of the class `DiscountSale`. Yet, in the method call `d1.lessThan(d2)`, the line that calls the method `bill` knows enough to use the definition of the method `bill` given for the class `DiscountSale`. This all works out because Java uses late binding.

Display 8.3 gives a sample program that illustrates how the late binding of the method `bill` and the methods that use `bill` work in a complete program.

Display 8.4 No Late Binding with Static Methods ★

```
1  /**
2   Demonstrates that static methods use static binding.
3  */
4  public class StaticMethodsDemo
5  {
6      public static void main(String[] args)
7      {
8          Sale.announcement();
9          DiscountSale.announcement();
10         System.out.println(
11              "That showed that you can override a static method " +
                "definition.");

12             Sale s = new Sale();
13             DiscountSale discount = new DiscountSale();
14             s.announcement();
15             discount.announcement();
16             System.out.println("No surprises so far, but wait.");

17             Sale discount2 = discount;
18             System.out.println(
19                   "discount2 is a DiscountSale object in a Sale variable.");
20             System.out.println("Which definition of announcement() will " +
                  "it use?");
21             discount2.announcement();
22             System.out.println(
23                   "It used the Sale version of announcement()!");
24     }
25 }
```

Java uses static binding with static methods so the choice of which definition of a static method to use is determined by the type of the variable, not by the object.

`discount` and `discount2` name the same object, but one is a variable of type `Sale` and one is a variable of type `DiscountSale`.

Sample Dialogue

```
This is the Sale class.
This is the DiscountSale class.
That showed that you can override a static method definition.
This is the Sale class.
This is the DiscountSale class.
No surprises so far, but wait.
discount2 is a DiscountSale object in a Sale variable.
Which definition of announcement() will it use?
This is the Sale class.
It used the Sale version of announcement()!
```

If Java had used late binding with static methods, then this would have been the other announcement.