Special Topic 10.3

Protected Access

We ran into a hurdle when trying to implement the `deposit` method of the `CheckingAccount` class. That method needed access to the `balance` instance variable of the superclass. Our remedy was to use the appropriate method of the superclass to set the balance.

Java offers another solution to this problem. The superclass can declare an instance variable as *protected:*

```
public class BankAccount
{
   . . .
   protected double balance;
}
```

Protected features can be accessed by all subclasses and by all classes in the same package.

Protected data in an object can be accessed by the methods of the object's class and all its subclasses. For example, `CheckingAccount` inherits from `BankAccount`, so its methods can access the protected instance variables of the `BankAccount` class. Furthermore, protected data can be accessed by all methods of classes in the same package.

Some programmers like the `protected` access feature because it seems to strike a balance between absolute protection (making all instance variables private) and no protection at all (making all instance variables public). However, experience has shown that protected instance variables are subject to the same kinds of problems as public instance variables. The designer of the superclass has no control over the authors of subclasses. Any of the subclass methods can corrupt the superclass data. Furthermore, classes with protected instance variables are hard to modify. Even if the author of the superclass would like to change the data implementation, the protected variables cannot be changed, because someone somewhere out there might have written a subclass whose code depends on them.

In Java, protected instance variables have another drawback—they are accessible not just by subclasses, but also by other classes in the same package (see Special Topic 8.9).

It is best to leave all data private. If you want to grant access to the data to subclass methods only, consider making the *accessor* method protected.