Programming Assignment 4-3

In this lab, we introduce some refinements to the requirements for Lab 3-2 that will lead to an application of polymorphism. The starting point for this lab is therefore the code you wrote for Lab 3-2.

We now add the following 3 requirements for managing `Account`s:

- when `balance` is read for checking account, a $5 monthly service charge will be subtracted

- when a `withdrawal` is made from a retirement account, a 2% penalty is applied to the balance

- when `balance` is read for savings, a 0.25% monthly interest rate is applied

Now, the algorithms for performing withdrawals, deposits, and reading balance differ (in some cases) among the different types of accounts. So you will implement these new requirements by introducing three subclasses of `Account`: `CheckingAccount`, `SavingsAccount`, and `RetirementAccount`.

With these new subclasses, you will also introduce some improvements in the code for Lab 3-2. Here are areas that need improvement together with suggested improvements:

(1) Each of the new subclasses will keep track of its own account type; however, you will not need to store this value as an instance variable any longer. Just provide the relevant value in the `getAcctType()` method of each subclass. For example, in `CheckingAccount`:

```
public AccountType getAcctType() {
   return Account.CHECKING;
}
```

As a result, you will need to modify the constructors in `Account` – they will no longer have an `acctType` argument.

(2) In Lab 3-2, `Account`s are listed separately in the `Employee` class; these should instead be stored in some kind of a list. This will allow you to add new types of `Accounts` in the (hypothetical) future without having to add new instance variables to `Employee` (recall the Open-Closed Principle).

```
//lab 3-2 code…
   public class Employee {
      private Account savingsAcct;
      private Account checkingAcct;
      private Account retirementAcct;
   ...
   }

//Replace with

   public class Employee {
      private AccountList accounts;
   ...
   }
```

An `AccountList` can be created by modifying your class `MyStringList` from Lab 3-3 to `AccountList`, where, instead of storing `String`s, your new class stores `Account` objects (actually, *references* to `Account` objects). Please do not use any of Java's List classes for this exercise.

(3) Since we do not wish to store different account types in separate instance variables (since you will be storing accounts in a list, as above), the `createXXX` methods need to be modified:

```
//lab 3-2 code…
public void createNewSavings(double startBalance){
    savingsAcct = new Account(this,AccountType.SAVINGS,startBalance);
}
public void createNewChecking(double startBalance){
    checkingAcct = new Account(this,AccountType.CHECKING,startBalance);
}
public void createNewRetirement(double startBalance){
    retirementAcct = new Account(this,AccountType.RETIREMENT,startBalance);
}
```

Keep these methods, but change their implementation by adding the newly created `Account` objects to the `AccountList`. For example:

```
public void createNewSavings(double startBalance) {
   Account acct = new SavingsAccount(this,startBalance);
   //accounts is the name of the AccountList variable
   accounts.add(acct);
}
```

(4) The `deposit` method in `Employee` needs to be recoded to make use of the `AccountList`.

```
//lab 3-2 code…
public void deposit(AccountType acctType, double amt){
    switch(acctType){
        case CHECKING:
            checkingAcct.makeDeposit(amt);
            break;
        case SAVINGS:
            savingsAcct.makeDeposit(amt);
            break;
        case RETIREMENT:
            retirementAcct.makeDeposit(amt);
            break;
        default:
    }
}
```

Improve this by changing the signature of `deposit` to

```
deposit(int accountIndex, double amt)
```

The `accountIndex` represents a selection that the User of our application makes in choosing one of the accounts in which to make a deposit; the selected `accountIndex` will correspond to the `Account` instance stored in the `AccountList` inside an `Employee` object.

The deposit can then be accomplished by these lines:

```
Account selected = accounts.get(acctIndex);
selected.makeDeposit(amt);
```

(Notice the nice use of polymorphism here.)

(5) Similar changes should be made to the `Employee withdraw` method. If the withdrawal cannot be made (because of insufficient funds), a message should appear in the console indicating this fact.

(6) Implement the new rules for reading balances and making withdrawals (listed at the top of this lab) in the following way. Provide the standard behavior for these functions (reading balance just returns the balance; making withdrawal just deducts the amount from the balance) in the `Account` superclass. But in cases where one of the rules above requires further processing, implement the additional processing by overriding the `Account` method in the relevant subclass.

For example, here is the implementation of `getBalance` in `SavingsAccount`:

```
public double getBalance() {
    double baseBalance = super.getBalance();
    double interest = (0.25/100)*baseBalance;
    return baseBalance + interest;
}
```

(One more improvement: 0.25 should be represented as a constant in `SavingsAccount`)

(7) The `getFormattedAccountInfo` method of `Employee` was implemented by checking whether each type of `Account` was `null`, and if not, reading the account information from it, and accumulating the results into an output `String`.

Now, all this can be accomplished by looping through the `AccountList` and gathering the same information.

(8) In the `main` method, we now want to make this a more interesting console application.

When the application starts, the User should see:

```
A. See a report of all accounts.
B. Make a deposit.
C. Make a withdrawal.
Make a selection (A/B/C):
```

If A is selected, then output the formatted report that you generated for Prog3-2.

If B is selected, the User should then interact with the system as in the following:

```
A. See a report of all accounts.
B. Make a deposit.
C. Make a withdrawal.
Make a selection (A/B/C): B

0. Jim Daley
1. Bob Reuben
2. Susan Randolph
Select an employee: (type a number) 2

0. checking
1. savings
2. retirement
Select an account: (type a number) 1

Deposit amount: 300.00
```

After the deposit is made, the User should see:

```
$300.0 has been deposited in the
savings account of Susan Randolph
```

The indexed list of `Employee` names, as displayed above, should be obtained by reading the array of `Employees` (created in the `main` method, as in Lab 3-2) However, the `main` method should not have access to the actual `AccountList` stored in `Employee` (violation of encapsulation). Instead, the list of account types for a particular `Employee` object should be accessed by calling a new method on `Employee`, namely, `getNamesOfAccounts`, which will return a list of the

account types in the form of a list of `String`s. Use `MyStringList` to store these account types, as `String`s, and remember that every `enum` equips each of its instances with a `toString` method. (Again, do not use any of Java's list classes for this part of the lab.)

For instance, if the `AccountList` stores a checking account and a savings account, in that order, the list returned by `getNamesOfAccounts` would store the `String`s "checking", "savings".

The same sequence of prompts as above should occur if the User initially selects C instead of B.

Your New Classes In this assignment, here is the package structure:

```
Prog4_3
   MyStringList.java
   AccountList.java
   Main.java

Prog4_3.employeeinfo
   Account.java
   AccountType.java
   SavingsAccount.java
   RetirementAccount.java
   CheckingAccount.java
   Employee.java
   AccountList.java
```