5 Abstracting over Data Definitions; Abstracting over Data Types

5.1 Abstracting over Data Definitions.

A bank customer can have three different accounts: a checking account, a savings account, and a line of credit account.

- The customer can withdraw from a checking account any amount that will still leave the minimum balance in the account. The customer can withdraw all money from a savings account. The balance of the credit line represents the amount that the customer already borrowed against the credit line. The customer can withdraw any amount that does not make the balance exceed the credit limit.
- The customer can deposit money to the account in any amount. If the customer deposits more to the credit line than the current balance, the balance could become negative — indicating overpayment. This is not allowed.

Review of Designing Methods for Unions of Classes.

The code in *lab5-banking.zip* defines the classes that represent this information. We will start using the Java convention of defining each class or interface in the file with the same name followed by the `.java` extension. That means, you need to build a project *Banking* and add to it the files *IAccount.java, Checking.java, Savings.java, Credit.java, and Examples.java*.

1. Make additional examples of data for these classes, then make sure you understand the rules for withdrawals.

   We have already defined the following the methods that manage the banking records:

   - The method `withdraw` that allows the customer to withdraw some desired amount.
   - The method `deposit` that allows the customer to deposit a given amount of money into the account.

2. Unfortunately, we are missing some test cases. Look at how we define tests that verify that the method throws the expected exception. Add a similar test for the method `withdraw` in the class `Credit`. Make

sure you understand the difference between withdrawing money from a credit line (i.e., increasing the loan balance) and withdrawing money from a checking or savings account (decreasing the remaining balance).

3. Strangely, we can expect the method `deposit` to throw an exception when invoked by a `Credit` account. Make sure you understand why this happens.

   Hint: Again, make sure you understand the difference between depositing money into a credit line (i.e., paying down the loan and decreasing the remaining balance), and depositing money into a checking or savings account (increasing the available balance).

   Now add a test that verifies that the method `deposit` throws the exception correctly in the class `Credit`.

5.2 Abstracting over Data Definitions: Lifting Fields

Look at the code and identify all places where the code repeats — the opportunity for abstraction.

Lift the common fields to an abstract class `Account`. Make sure you include a constructor in the abstract class, and change the constructors in the derived classes accordingly. Run the program and make sure all test cases work as before.

5.3 Abstracting over Data Definitions: Lifting Methods

For each method that is defined in all three classes decide to which category it belongs:

- The method bodies in the different classes are all different, and so the method has to be declared as `abstract` in the `abstract` class.
- The method bodies are the same in all classes and it can be implemented concretely in the `abstract` class.
- The methods look very similar, but each produces a different variant of the union — therefore it cannot be lifted to the `super` class.
- The method bodies are the same for two or more of the classes, but are different in one class — therefore we can define the common body in the `abstract` class and *override* it in only one derived class.

Design the following methods, keeping in mind the rules given above. When in doubt, design the method in each class separately, then see what rules apply.

1. Design the method `amtAvailable` that determines the amount of money that is available to withdraw.

2. Design the method `canWithdraw` that determines whether the customer can withdraw the desired amount from the account.

3. See what changes can you make to the `withdraw` method.

   - Make the exception report the name on the account when the exception indicates that a withdrawal is not possible.
   - Explain why we could not do this before defining the abstract class? (We could, but it would involve extra work.)
   - Explain why we cannot define a concrete method `withdraw` in the `abstract` class `Account`.

5.4 Understanding Equality

Note: This material is covered in pages 321 - 330 in the textbook. Read it carefully.

We now want to define a method that will determine whether the given account is the same as the given account. We may need such method to find the desired account in a list of accounts.

Of course, now that we have the abstract class it would be easy to compare just account number and the name on the account. But, maybe, we want to make sure that the customer's data match the data we have on file exactly - including the balances, the interest rates, and the minimum balances - as applicable.

The design of the method `same` is similar to the technique described in the textbook. The relevant classes and examples that were handed out in the class can be found in the file *Coffee.java*. You may want to look at the code there as you work through this problem.

1. Begin by designing the method `same` in the interface `IAccount`. This implies that the method needs to be defined in the abstract class `Account` either as an abstract method or as a concrete method.

2. Make examples that compare all kinds of accounts - both of the same kind and of the different kinds. For the accounts of the same kind you need both the expected `true` answer and the expected `false` answer. Comparing any checking account with another savings account must produce `false`.

3. Now that you have sufficient examples, follow with the design of the `same` method in one of the concrete account classes (for example the `Checking` class). Write the template and think of what data and methods are available to us.

4. You will need a helper method that determines whether the given account is a Checking account. So, design the method `isChecking` that determines whether this account is a checking account. You need to design this method for the whole class hierarchy - the interface `IAccount`, the abstract class `Account` and all subclasses. Do the same to define the methods `isSavings` and `isCredit`.

5. We are not done. This helps with the first part of the `same` method. We need another helper method that tells Java that our account is of the specific type. Here is the method header and purpose for the checking account case:

   ```
   // produce a checking account from this account
   Checking toChecking();
   ```

   In the class `Checking` the body will be just

   ```
   // produce a checking account from this account
   Checking toChecking(){
     return this; }
   ```

   Of course, we cannot convert other accounts into checking account, and so the method should throw a `RuntimeException` with the appropriate message. We need the same kind of method for every class that extends the `Account` class or implements the `IAccount`.

6. Finally, we can define the body of the `same` method in the class `Checking`:

   ```
   // produce a checking account from this account
   boolean same(IAccount that){
   ```

```
    if (that.isChecking()){
      return that.toChecking().sameChecking(Checking other);
    } else {
      return false;
    }
  }
```

That means, we still need the method `sameChecking` but this only needs to be defined within the `Checking` class and can be defined with a `private` visibility.

Finish this - with appropriate test cases.

7. Finish designing the `same` method for the other two account classes.

Note: The method above can be written with two Java language *features*, the `instanceof` operator and *casting* as follows:

```
// produce a checking account from this account
boolean same(IAccount that){
  if (that instanceof Checking){
    return ((Checking)that).sameChecking(Checking other);
  } else {
    return false;
  }
}
```

However, this version is problematic. If the class `PremiumChecking` extends `Checking`, then any object constructed with a `PremiumChecking` constructor will be an instance of `Checking` and the trouble that can result is illustrated in the example *TestSame.java*. You can make a simple project and run the examples, but we include the output from the *tester* for illustration.