**Programming Project 1**

**Due Friday, October 2 at 11:59pm**

This first programming assignment is to give you practice writing classes, methods, fields and if statements as well as practice creating objects and calling methods. Loops are not needed. You should not use loops for the classes below, but you may use them as part of the testing report. All your classes should extend Object.

In this project, you will be writing the necessary classes and methods to implement many of the features of a bank account.

***IMPORTANT:****Read the****Do's and Dont's****in the****Course Honor Policy****found on the Canvas Pages.*

**I. Code Readability (20% of your project grade)**

Once upon a time, getting a program to work was the only goal of programming. That was before computers took over the world. Now, with highly complex software running much of our lives, the industry has learned that computer code is a written document that must be able to communicate to other humans what the code is doing. If the program is too hard for a human to quickly understand, the industry does not want it.

Most companies enforce readable code by having very strict rules about how the program should look. This class will do the same but will not be quite as strict so you can have some freedom for developing your own style.

**To receive the full readability marks, your code must follow the following guideline:**

* All variables (fields, parameters, local variables) must be given appropriate and descriptive names.
* All variable and method names must start with a lowercase letter. All class names must start with an uppercase letter.
* The class body should be organized so that all the fields are at the top of the file, the constructors are next, and then the rest of the methods.
* Every statement of the program should be on its own line and not sharing a line with another statement.
* All code must be properly indented (see Appendix F of the Lewis book for an example of good style). The amount of indentation is up to you, but it should be at least 2 spaces, and it must be used consistently throughout the code.
* You must be consistent in your use of {, }. The closing } must be on its own line and indented the same amount as the line containing the opening {.
* There must be an empty line between each method.
* There must be a space separating each operator from its operands as well as a space after each comma.
* There must be a comment at the top of the file that includes both your name and a description of what the class represents.
* There must be a comment directly above each method that, in one or two lines, states *what* task the method is doing, not how it is doing it. Do not directly copy the homework instructions.
* There must be a comment directly above each field that, in one line, states what the field is storing.
* There must be a comment either above or to the right of each non-field variable indicating what the variable is storing. Any comments placed to the right should be aligned so they start on the same column.

**II. Program Testing Document (20% of your project grade)**

Once upon a time, companies thought errors in code were only a minor inconvenience. That was before software glitches started killing people and destroying companies. Now, standard practice is that all code must be thoroughly verified before a company is willing to release it. At a large number of firms, programmers are required to *first* design the test cases the program must pass and *then* start writing the code. We will not be that strict in this class, but you will need to test your code.

**To receive full testing marks, you must write a testing report that shows that you thoroughly tested every method of the program.** The report should be a short English description for each test (what you are testing and what the expected result of the test is) followed by the actual result of the test. If you are using DrJava, you can enter the test into the interactions pane and then copy and paste the test code plus the result to your report. If you fail to complete the program, your report should indicate how you would go about testing the incomplete methods.

**Your grade on the testing report is how thoroughly you test your code, not how correctly your code runs.** If your code is not 100% correct then your report *should* show an incorrect result to some test. Testing methods that do not have conditional statements should be pretty straightforward, but you need to put thought into testing methods with conditional statements so that each branch of the if-statement is tested.

*Hint 1*: You can test multiple methods with one test. For example, you can test each setter/getter method pair together or you can test constructors and getter methods together.

***Hint 2: Do not put off testing to the end!*** Test each method after you complete it. Many methods depend on other methods. Delaying testing could mean cascading errors that cause your whole project to collapse. Since you need to test anyway, copy the tests you do into a document, and you are most of the way to completing your report.

If you are not using DrJava, you are allowed (but not required) create a separate class that tests your program. You must still write a testing report that documents the tests you do in this class. Do not place testing code into a main method of the classes below. That is not the purpose of a main method.

**III: Programming (60% of your grade will be program correctness)**

Create the following four classes:

1. BankAccount  
   The BankAccount class will need instance fields to keep track of the current balance, the minimum balance that should be maintained in the account, the interest rate, the ATM fee, the overdraft fee, and more. The class will have the following methods:

The BankAccount class should have two constructors:

* 1. A generic constructor that takes no values. This constructor lets you use the account with all the fields set to their default values.
  2. A constructor that takes five inputs.

The first is a double and is the interest rate for the account,

the second is a int as is the minimum balance for the account,

the third is a double and is the overdraft fee for the account,

the fourth is a double and is the ATM fee for the account, and

the fifth is a double and is the bounced check fee for the account.

* 1. deposit  
     Takes a single double as input and adds this value to the account's balance. There is no return value.
  2. getBalance  
     Takes no input and returns a double. Returns the current balance for the account.
  3. getInterestRate and setInterestRate  
     The interest rate is a double. The interest rate should be stored as an appropriately named and typed variable.
  4. getMinimumBalance and setMinimumBalance  
     The minimum balance threshold for the account is an int because the minimum balance is in whole dollars. Use an appropriately named variable to store this value. The minimum balance is a threshold, **and if the balance of the account falls below this value, a fee is charged to the account.**
  5. getATMFee and setATMFee  
     The ATM fee is a double, and it is a fee that is charged whenever **the withdrawATM method successfully withdraws money from the account.**
  6. getOverDraftFee and setOverdraftFee  
     The overdraft fee is a double and it is a fee that is charged **if, at the end of the day, the account's balance is less than the account's minimum balance.** **The fee is only charged once per month.**
  7. getBouncedCheckFee and setBouncedCheckFee  
     The bounced check fee is a double and the fee **is charged if the withdrawDraft method attempts to remove more money than is available in the account's balance.**
  8. getWithdrawFee and setWithdrawFee  
     The withdraw fee is a double and is a fee charged per withdraw **when the user has exceeded the maximum number of monthly withdraws allowed on the account.**
  9. getWithdrawLimit and setWithdrawLimit  
     The withdraw limit is an int value. The limit is 0 if the account allows an unlimited number of withdraws. Otherwise, the limit is the maximum number of free withdrawals allowed per month.
  10. withdraw  
      Takes a single double as input. If the current balance is greater or equal to the input amount, then method returns true after the balance is reduced by this amount, the total number of withdrawals for the month is incremented, and the withdraw fee is applied (if the total number of withdrawals exceeds the limit). Otherwise, the method returns false and nothing is removed from the account's balance.
  11. withdrawDraft  
      Takes a single double as an input. If the current balance is greater or equal to this amount, then the balance is reduced by the amount, the method returns true, the total number of withdrawals for the month is incremented, and the withdraw fee is applied (if the total number of withdrawals exceeds the limit). Otherwise, the balance is reduced by the bounced check fee amount and the method returns false.
  12. withdrawATM  
      Takes a single double as an input amount. If the current balance is greater or equal to the sum of this amount and the ATM fee, then the balance is reduced by this amount, the balance is further reduced by the ATM fee, the method returns true, the total number of withdrawals for the month is incremented, and the withdraw fee is applied (if the total number of withdrawals exceeds the limit). Otherwise, the method returns false and nothing is removed from the account's balance. *Hint:* You can write withdrawATM with very little code if you call the withdraw method appropriately.
  13. incrementDay  
      This method takes no input and returns nothing. You will need to add two variables to your class. One variable will store the interest earned so far this month. Accounts with an interest rate earn interest daily, but the interest is only credited to the account monthly. The incrementDay method updates the amount that the account has earned so far.  
      You will also need a boolean variable that stores whether the overdraft fee has been charged this month. If the account balance falls below the minimum balance, the overdraft fee should be charged, but the fee should only be charged once per month. This variable will be called the *overdraft flag* and the variable will be true if the fee has been charged and false if the fee has not been charged.  
      The method incrementDay should do the following.
      1. If the account balance is below the account minimum balance and the overdraft flag is false, then the balance is reduced by the overdraft fee, the overdraft flag is set to true, and no interest is earned.
      2. If the account balance is below the account minimum balance and the overdraft flag is true, no interest is earned and nothing is deducted from the balance.
      3. Otherwise, compute the interest earned that day by
         1. summing the current balance and any interest earned so far this month,
         2. multiplying this sum by the interest rate divided by 365 (this must be done appropriately),
         3. and add then adding this computed interest to the interest earned so far.
  14. incrementMonth  
      This method takes no input and returns nothing. The incrementMonth function performs the following tasks.
      1. The interest earned so far is added to the account balance.
      2. The interest earned so far is reset to 0.
      3. The overdraft flag is set to false.

1. CreditCardAccount  
   The CreditCardAccount will have variables to store the current balance, the credit limit, the interest rate, and more. The class will have the following methods:

The CreditCardAccount class should have two constructors:

* 1. A generic constructor that takes no input and lets you use the credit card account with all fields set to their default values.
  2. A constructor that takes four inputs.

The first is an int and is the credit limit.

The second is a double and is the interest rate.

The third is an int and is the minimum payment for each month, and

the fourth is an int and is the late payment penalty.

* 1. getCreditLimit and setCreditLimit  
     The credit limit is the maximum amount that may be borrowed. It is an int because it is in whole dollars. This value should be stored in an appropriately named variable.
  2. getInterestRate and setInterestRate  
     The interest rate for the account. The rate is a double.
  3. getMinMonthlyPayment and setMinMonthlyPayment  
     Sets the minimum amount that must be paid to avoid a late payment penalty. The value is an int.
  4. getLatePaymentPenalty and setLatePaymentPenalty  
     The late payment penalty is a penalty charged on a late payment. The value is an int.
  5. getBalance  
     Takes no input and returns the current balance for the account as a double.
  6. getMonthlyPayment  
     Takes no input and returns a double which represents the amount that the user needs to pay this month in order to pay off the credit card in full. This value is different than the balance, and the class should have an appropriately named variable for this value.
  7. charge  
     This method takes a single double amount as input. If the sum of the current balance and the amount that was input is less than or equal to the credit limit, add the amount to the current balance and return true. Otherwise add nothing to the current balance and return false.
  8. payment  
     This method takes a single double value and returns nothing. Subtract the input value from the current balance and add the input value to a variable that stores the total amount paid so far this month.
  9. incrementDay  
     This method takes no input and returns nothing. If the previous month's balance was not paid in full, the credit card charges interest on the current balance after each day. To do this, the class will need two additional fields, a variable that keeps track of the interest charged so far and a boolean variable that is a paid-in-full flag that keeps track of whether the user has paid the balance in full.  
     incrementDay works as follows:
     1. If the paid-in-full flag is false then
        1. sum the account balance and the interest charged so far,
        2. then multiply this sum by the interest rate divided by 365,
        3. and add this value to the interest charged so far.
     2. If the paid-in-full flag is true, then there is nothing to do.
  10. incrementMonth  
      This method takes no input and returns nothing.
      1. Add the interest charged so far to the account balance.
      2. Reset the interest charged so far to 0.
      3. If the total payment this month is greater than or equal to the monthly payment, set the paid-in-full flag to true, otherwise set the flag to false.
      4. If the minimum monthly payment is smaller than the monthly payment and the total payment this month is less than the minimum monthly payment, add the late payment penalty to the current balance.
      5. Set the total payment this month to 0.
      6. Set the monthly payment equal to the current balance.

1. Account  
   The Account class will have numerous fields and the following methods:

The Account class should have one constructor:

* 1. A constructor that takes five inputs:

two Strings representing the first and last name for the account, a String representing the street address,

a String representing the zipcode, and

a Date.

* 1. getFirstName and setFirstName  
     Gets and sets the first name on the account as a String.
  2. getLastName and setLastName  
     Gets and sets the last name for the account as a String.
  3. getStreetAddress and setStreetAddress  
     The street address (for example, house number and street name) for the account. The street address should be stored as a single String variable.
  4. getZipCode and setZipCode  
     The zip code for the account's address, stored as a String.
  5. setDate and getDate  
     Sets and returns a date associated with the account. The type of the date should be the Date class created below.
  6. getSavingsAccount and setSavingsAccount  
     The Account class should have an appropriately named field of type BankAccount that represents the savings account. The method will assign and retrieve the value of this field. (Note: the method will not create an account but will assume that the parameter is a correctly created account. Also, note that there really is no distinction between different kinds of accounts. They are all BankAccount objects.)
  7. getCheckingAccount and setCheckingAccount  
     The checking account is also a BankAccount, similar to the savings account.
  8. getMoneyMarketAccount and setMoneyMarketAccount  
     The money market account is also a BankAccount.
  9. getCreditCardAccount and setCreditCardAccount  
     We will assume there can only be one credit card account per account. The type is CreditCardAccount.
  10. incrementDay  
      This method takes no input and returns nothing. incrementDay should do each of the following tasks in order:
      1. Call the incrementDay method on the account's associated date.
      2. For each account that is not null, call the account's incrementDay method.
      3. If the result of calling the date's incrementDay method resulted in changing the month of the date, then for each account that is not null, call the account's incrementMonth method.
  11. equals Change the inherited method equals so that two accounts are considered equal if they have the same name (first and last), address, and zipcode.
  12. toString Change the inherited toString method so that the string representation of the account is *name*, *address*, *zipcode* and then for each account that exists, the type of account and the balance of that account. For example, if an account has a savings account and a credit card account but no money market or checking account, the method should return a string similar to: *"Harold Connamacher 502 Olin 44106. Savings: 1320.0, Credit Card: 516.3"*.

1. Date  
   The Date class will represent a date. (Java has both Date and Calendar in the API, but both are more complex than needed for this homework.) The Date class should have the following methods:

The Date class should have one constructor:

It has input int day,

int month and initializes the Date object with the given inputs.

You may assume that the inputs are all valid values.

* 1. getDay: returns the day of the date. The day should be a value between 1 and 31.
  2. getMonth: returns the month of the date. The month should be between 1 and 12.
  3. incrementDay: adds 1 to the day of the date. If the day exceeds the number of days for the month, the day is set to 1 and the month is incremented. (Note that we will not be keeping track of years and so you should ignore leap years.) If the month exceeds 12, the month is set to 1.
  4. equals: Override the inherited equals method so that two dates are equal if they have the same day and month.
  5. toString: Override the inherited toString method so that the string representation of the date is something nice involving the month and day. The exact format is up to you.

**iV. Submitting Your Project**

Submit the .java files (not the .class files or the .java~ files) for each of your classes plus the testing report on Canvas.