Project #2 – 130 points

Due Date

Monday, February 28, 11:59pm.

Submission

- Submit **a single zipped file** to Canvas. The zipped file MUST include the following grading items.
  (1) Source folder **src** [80 points]
    (a) Create a package to organize the source files *.java; use all lowercase letters for the package name
    (b) MUST include all team members' names using the **@author** tag in the comment block on top of EVERY Java class, or you will **lose 2 points**.
  (2) Class Diagram [10 points]
  (3) **JUnit** test classes
    (a) Date class, test isValid() method. [10 points]
    (b) MoneyMarket class, test monthlyInterest() [5 points]
    (c) AccountDatabase class, test open() and close() methods [20 points]
  (4) **Javadoc** folder **doc**, including all the files generated. [5 points]
- The submission button on Canvas will disappear after **February 28, 11:59pm**. It is your responsibility to ensure your Internet connection is good for the submission. **You get 0 points** if you do not have a submission on Canvas. DO NOT wait until the last minute. DO NOT send the project to me or the graders through the emails.

Project Description

Your team will develop a software that behaves like a bank teller to process banking transactions. This project uses the IDE console as the user interface and utilizes the Java standard input and output to read the transactions and write the results. The transactions may include, open a new account, close an existing account, deposit money to an existing account, withdraw money from an existing account, and print the account details. The software system maintains an account database, which may include 4 different types of bank accounts listed in the table below. Same person can hold different types of accounts. The interest rates and fee schedules are different based on the account types and account options.

Account Type	Monthly Fee / waived if balance is >=	Annual Interest Rate	Account Options
Regular Checking	$25 / $1000	0.1%	N/A
College Checking	$0	0.25%	Provided to Rutgers's students only. Must provide a valid campus code to qualify: 0 – New Brunswick 1 – Newark 2 – Camden
Savings	$6 / $300	0.3%	A loyal customer's account gets additional interest rate 0.15%; that is, annual interest rate will be 0.45% for a loyal customer account.
Money Market	$10 / $2500	0.8%	• By default, it is a loyal customer account • If the balance falls below $2500, then it is not a loyal customer account anymore • A loyal customer account gets additional interest rate 0.15%; that is, annual interest rate will be 0.95%. • Fee cannot be waived if the number of withdrawals exceeds 3 times

A transaction is a command line that always begins with a command, in uppercase letters, and followed by several data tokens delimited by spaces. **Commands are case-sensitive**, which means the commands with lowercase letters are invalid. You are required to deal with bad commands not supported, or you will **lose 2 points** for each bad command not handled. Your software must support the following commands.

- **O** command, to **open an account** with the desired account type. There are 4 account types: C – regular checking, CC – college checking, S – savings, MM – money market savings. Account holders are identified by their profiles. Each profile includes the account holder's first name, last name, and date of birth. **Names are NOT case-sensitive**. Initial deposit is required to open a new account. When an account is opened, the account is added to the account database. Below is a list of sample transactions for opening an account. It is possible that the user didn't enter enough information for opening an account. You must handle the exceptions.

```
O  C  John Doe 2/19/1989
O  CC Jane Doe 10/1/1995 599.99 0
O  S  April March 1/15/1987 1500 1
O  MM Roy Brooks 10/31/1979 2909.10
```

  The above transaction starts with the O command followed by the account type, first name, last name, date of birth, and initial deposit. College checking is provided only to Rutgers's students and must provide the campus code: 0 – New Brunswick, 1 – Newark, or 2 – Camden. It is possible the user enters an invalid campus code. If an account holder has a Checking account, the account holder cannot open a College Checking, as each account holder can only have one Checking account. Savings accounts get special interest rate and must provide the code: 0 – non-loyal customer, or 1 – loyal customer. Money market accounts need an initial deposit at least $2,500 to open and are set to loyal customer accounts by default. You should not open the account if the date of birth is not a valid calendar date, or it is today or a future date. Zero or negative amounts for the initial deposit should be rejected. O command can also be used to reopen a closed account. In this case, set the account to open again, and update the account with the new information provided.

- **C** command, to **close an account**, for example,

```
C  MM   Jane Doe 10/1/1995
C  C    April March 1/15/1987
C  CC   John Doe 2/19/1989
C  S    April March 1/15/1987
```

  The above transaction set the account to close, set the balance to 0, and set the is loyal account to false. It doesn't reset the campus code for College Checking. Closed accounts remain in the database and can be re-open later with new deposit or campus code where applicable.

- **D** command, to **deposit money** to an existing account. You should reject the transaction if a negative amount or 0 is entered. Below are some sample transactions.

```
D  C    John Doe 2/19/1990 100
D  CC   Kate Lindsey 8/31/2001 100
D  MM   Roy Brooks 10/31/1979 100.99
D  S     April March 1/15/1987 100
```

- **W** command, to **withdraw money** from an existing account. Command line formats are the same with the D command, and sample rules apply. In addition, you must check if there is enough balance for the withdrawal.

- **P** command, to **display all the accounts in the database,** with the current order in the array**.** You should add "CLOSED", for the accounts that are closed. Account balances should be displayed with 2 decimal places, see the sample output for the format.
- **PT** command to **display all the accounts in the database,** ordered by the account type. For the same account type, the order doesn't matter. You should add "CLOSED", for the accounts that are closed.
- **PI** command, to **display all the accounts in the database** with calculated fees and monthly interests based on current balances. Fees and interests should be displays with 2 decimal places, see the sample output for the format.
- **UB** command, to **update the balances for all accounts** with the calculated fees and monthly interests and **display all the accounts in the database** with the updated balances.
- **Q** command, to stop the program execution and display `"Bank Teller is terminated."`

Project Requirement

1. You MUST follow the Software Projects Coding Standard and Ground Rules posted on Canvas under "Modules" "Week #1". You will lose points if you are not following the rules.
2. The text file "**project2_testCases.txt**" posted on Cavas contains the test cases the graders will be using to test your projects. You should match the output in "**project2_sampleOutput.txt**". Your project should be able to take all the test cases in batch with the same sequence in the text file without getting any exceptions and without terminating abnormally, or you **will lose 5 points**. You will **lose 2 points** for each incorrect output or each exception.
3. Each Java class must go in a separate file. **-2 points** if you put more than one Java class into a file.
4. Your program MUST handle bad commands! **-2 points** for each bad command not handled.
5. You are NOT allowed to use any Java library classes, EXCEPT **Scanner**, **StringTokenizer, Calendar**, **DecimalForamt**, **Java wrapper classes**, and necessary **Java Exception classes**. **You will lose 5 points** for each additional Java library class imported, with a **maximum of losing 10 points**.
6. You are NOT allowed to use any Java Collections classes, such as **ArrayList** or **LinkedList**, or you **will get 0 points for this project**.
7. When you import Java library classes, be specific and DO NOT import unnecessary classes or import the whole package. For example, `import java.util.*;`, this will import all classes in the `java.util` package. You will **lose 2 points** for using the asterisk "*" to include all the Java classes in the `java.util` package, or other java packages, with a **maximum of losing 4 points.**
8. You MUST create a **Class Diagram** for this project to document the software structure. The diagram is worth 10 points. You will **lose the 10 points** if you submit a hand-drawing diagram.
9. You are not allowed to have redundant code in this project; **reuse the code in the superclass as much as possible** by using the **super keywords**. **-2 points** for each violation below.
    - Unused code: you write code that was never called or used.
    - Duplicate code segments (repeat the same code segment) for the same purpose in more than one place/class.
    - Define common instance variables in each of the subclasses.
    - Define specific instance variables in the superclass that are not used by all the subclasses.
    - Define common methods in each subclass instead of defining it in the superclass.
10. You must define classes with the following inheritance relationships. **-5 points** for each class missing, or incorrect inheritance structure.
    - **Account** class is **an abstract class** that defines the common data and operations for all account type; each account has a profile that uniquely identifies the account holder. This is **the superclass of all account types**, and it is an abstract class with 3 abstract methods. You must implement and use the methods listed

below and you CANNOT add additional instances variable or change the signatures of the methods. **-2 points** for each violation. You can add additional methods if necessary.

```
public abstract class Account {
     protected Profile holder;
     protected boolean closed;
     protected double  balance;

     @Override
     public boolean equals(Object obj) { }
     @Override
     public String toString() { }

     public void withdraw(double amount) { }
     public void deposit(double amount) { }
     public abstract double monthlyInterest(); //return the monthly interest
     public abstract double fee();         //return the monthly fee
     public abstract String getType(); //return the account type (class name)
}
```

- **Checking** class extends the Account class and includes specific data and operations to a regular checking account.
- **CollegeChecking** class extends the Checking class and includes specific data and operations to a college checking account.
- **Savings** class extends the Account class and includes specific data and operations to a savings account.
- **MoneyMarket** class extends the Savings class and includes specific data and operations to a money market account.

11. **Polymorphism is required**. You CANNOT use the getClass() method for checking the class type listed in #10 above, or you will **lose 10 points.** Implement the equals() methods where necessary and call the right equals() method to compare the account instances. If you are overriding any methods, you must add the annotation **@Override** on top of the methods. **-2 points** for each violation.
12. In addition to the classes in #10, you must include the classes listed below. **-5 points** for each class missing. You CANNOT perform I/O (**System.in** and **System.out**) in all classes, EXCEPT the BankTeller class, and the 3 print methods in the AccontDatabase class. **-2 points for each violation.** The floating-point numbers must be displayed with 2 decimal places. **-1 point for each violation.**
    - **Date class.** Import this class from your Project 1.
    - **Profile class.** Define the profile of an account holder as follows. You cannot add additional instance variables. **-2 points** for each violation. You should define equals() and toString() methods.

```
public class Profile {
    private String fname;
    private String lname;
    private Date   dob;
}
```

    - **AccountDatabase class.** An instance of this class is an array-based container that holds a list of accounts with different types. The initial capacity of the container is 4. It will automatically grow the capacity by 4 if the array is full. The array-based container doesn't shrink in capacity. You must implement and use the methods listed and you CANNOT add additional instances variable or change the signatures of the methods. **-2 points** for each violation. You can add additional methods if necessary, however, all the **public methods** you defined must take either a single parameter (Account account), or no parameters, **-2 points** for each violation.

```
public class AccountDatabase {
    private Account [] accounts;
    private int numAcct;

    private int find(Account account) { }
    private void grow() { }
    public boolean open(Account account) { }
    public boolean close(Account account) { }
    public void deposit(Account account) { }
    public boolean withdraw(Account account) { } //return false if insufficient fund
    public void print() { }
    public void printByAccountType() { }
    public void printFeeAndInterest() { }
}
```

- **BankTeller class.** This is the **user interface class** that performs read/write through the console (standard input and output.) You should define a run() method to handle the transactions, and keep the run() method under 35 lines, not including the braces, or **lose 3 points**. This class **handles all exceptions and invalid data** before it calls the methods in AccountDatabase class to complete the associated transactions. For example, you could get **InputMismatchException, NumberFormatException**, **NoSuchElementException,** invalid dates, invalid campus codes, 0 and negative amounts for deposit or withdraw money. Whenever there is an exception or invalid data, display a message to the console. **-2 points** for each exception not caught or invalid data not checked in this class or messages not displayed. See the sample output for error messages.

13. **JUnit test.** Design the test cases and implement the test cases with JUnit 4 or JUnit 5 to test the following methods. Generate a test class for each of the classes you are testing. As a result, you will have **3 test classes**.
    - **isValid()** methods in Date class.
    - **monthlyInterest()** in MoneyMarket class
    - **open() and close()** methods in AccountDatabase class
14. You are required to **generate the Javadoc** after you properly commented your code. Your Javadoc must include the documentations for the constructors, private methods, and public methods of all Java classes. Generate the Javadoc in a single folder "**doc**" and include it in the zip file to be submitted to Canvas. **Please double check your Javadoc after you generated it** and make sure the descriptions are NOT EMPTY. You will lose points if any description in the Javadoc is empty. You will **lose 5 points** for not including the Javadoc.