BTP400 Assignment #1 (Winter 2020)

Due: 11:58 pm, February 22 (Saturday), Week 7
Late Penalty: -10% per day (including weekends)

Learning Objectives

1. **Demonstrate the ability to build a Java application according to user requirements.**
2. **Demonstrate the mastery of fundamental object-oriented programming skills in Java. In particular, Java interface, inheritance and polymorphism.**
3. **Do research reading on Java APIs (BigDecimal and StringBuffer).**
4. **Demonstrate the use of Java packages to organize Java classes.**
5. **Demonstrate the use of command lines to extract a JAR file, compile and execute Java packages.**
6. **Demonstrate the ability to create JUnit tests.**
7. **Demonstrate the use of Eclipse to run JUnit tests.**
8. **Demonstrate the use of javadoc comments.**
9. **Document the source code properly.**
10. **Develop coding style by using Java Coding Conventions.**

Overview

You are asked to develop a menu-based financial application that runs on a Java platform. You are going to modify the **Account** and **Bank** classes that you have developed in the lab problems. Furthermore, you are going to create two subclasses of the **Account** class (**Chequing** and **GIC**), the Taxable interface, the application class (**FinancialApp**) and JUnit tests.

Important Notes

1. This is a group assignment. You should form a team of two people as soon as possible. Give the names of the team members to the professor. (No last-minute team formation is allowed.) Each team member must contribute evenly to implement and test the Java classes and the Java interface. Otherwise some of your marks will be deducted according to the peer review. In particular, the implementation of the Java classes (Account,

Chequing, GIC and FinancialApp) should be shared evenly among the team members. If a team member is the author of a Java class under test (e.g. Account), another team member should be the author of the JUnit tests for that particular class. The authorship of each Java class must be documented clearly. (Note: Please ask the professor for permission if you want to complete this assignment individually.)

2. You will work with a team member to create a suite of JUnit tests. You must divide the work evenly. You must use the javadoc tags to document clearly the authorship of all classes under test and JUnit tests.

3. You must follow the basic guidelines of Java Coding Conventions. Otherwise you will lose 21% of your marks (B+ max).

4. You must document your source code properly (such as indentation, blank lines and comments). You must also use javadoc documents to document all classes. Otherwise you will lose 21% of your marks (B+ max).

5. You must use the **BigDecimal** class to do financial calculations. You should read the Java API and this article: http://www.javaworld.com/article/2075315/core-java/make-cents-with-bigdecimal.html

   Thus the data types of the (private) data fields that hold monetary values of an account (e.g. current balance and final balance) must be **BigDecimal**. However, the date types of the method parameters and the return types remain *double*.

6. You must use the **StringBuffer** class when you are concatenating a string. The return type of a toString( ) method is still String, however. The reason is performance concern: http://www.javaworld.com/article/2076072/build-ci-sdlc/stringbufferversus-string.html.

Part 1A: Specifications of an Inheritance Hierarchy and a Java Interface.

1.1 API of the Account Class

Account( )
Account(String fullName, String accountNumber, double balance)

String getFullName( )
String getFirstName( )
String getLastName( )
String getAccountNumber( )
double getBalance( )

String toString( )
int hashCode( )
boolean equals(Object obj)

boolean withdraw(double amount)
void deposit(double amount)

1.2 Functional Requirements

Refer to the functional requirements of the **Account** class developed in the lab activities. You will enhance the Account class to satisfy the following requirements.

Use a **BigDecimal** object to store the current balance.

You need to implement the withdraw( ) and deposit( ) methods that update the current balance of an account. The toString( ) method returns a string that uses the following format:

```
Name           : Smith, John
Number         : A0007
Current Balance: $1002.99
```

```
Note: Pay attention to the format(e.g. "Smith, John").
```

1.3 Constraints

The (current) balance of an account must never become negative.
If a constructor receives a *negative* balance, it will initialize the balance to zero. If the deposit( ) or withdraw( ) method receives a *negative* amount, it will not update the (current) balance. The withdraw( ) method will return false and the balance will not be updated if an account does not have enough money to complete the withdrawal.

Note: You must create JUnit tests to validate that all these constraints have been implemented properly. (See Part 4 for details.)

2.1 API of the Taxable Interface

```
void calculateTax( )
double getTaxAmount( )
String createTaxStatement( )
```

Note: The tax rate (in percentage) is a constant in the interface.

2.2 Functional Requirements

The calculateTax( ) method uses a tax rate to calculate the amount of tax on the basis of interest income. The getTaxAmount( ) method returns the amount of tax that has been calculated. The createTaxStatement( ) method returns a string that uses the following format:

```
Tax rate       : 15%
Account Number : D1234
Interest income: $500.00
Amount of tax  : $ 75.00

Note: Pay attention to the format (e.g. the leading blank
      spaces and the alignment of decimal points).
```

3.1 APIs of the Chequing and GIC Classes

Chequing and **GIC** are subclasses of the **Account** class. You should use the *getBalance( )* method in the **Account** class to return the current balance of an account.

3.1.1 The Chequing Class

It is not **Taxable** and has two constructors. When a **Chequing** object is created, it is provided with a full name (e.g. “John Doe”), an account number, the starting balance, service charge per transaction and maximum number of transactions allowed. The default constructor uses the default constructor of the **Account** class. It initializes service charge to $ 0.25 and the maximum number of transactions to 3. In your implementation, a **Chequing** object uses an **array** (not an ArrayList) to keep track of all the transactions (i.e. deposits and withdrawals) made on the account.

You must code the equals( ), toString( ) and hashCode( ) methods. The toString( ) method uses the toString( ) method of the **Account** class and includes information about the account type, service charge, total service charges, and a list of transaction amounts.

The toString( ) method returns a string that has the following format:

```
Name            : Ryan, Mary
Number          : B0003
Current Balance: $200.00
Account Type         : CHQ
Service Charge       : $0.25
Total Charges        : $0.75
List of Transactions: +50.00, -25.00, +30.00
Final Balance        : $199.25
```

Note: Pay attention to the format (e.g. “Ryan, Mary”). You need not align the decimal points.

You must override the deposit( ) and withdraw( ) methods of the **Account** class. These two methods store the amounts of transactions in the array.

(The amount of a withdrawal should be stored as a negative number.) Then they will update the current balance and the total amount of service charges if transactions are made successfully. However, they will not deduct the amount of service charges from the current balance.

A transaction will not be completed successfully if
i) negative numbers are passed into these methods,
ii) the maximum number of transactions will be exceeded, or
iii) the balance would become negative if the transaction were completed.

The withdraw( ) method will return false if the transaction cannot be completed.

You must override the *getBalance( )* method such that it returns the final balance of the account. The final balance is calculated by subtracting the total amount of service charges from the current balance. (The final balance will become negative if the current balance is zero and the service charges are greater than zero.)

3.1.2 The GIC Class

It is **Taxable** and has two constructors. When a **GIC** object is created, it is provided with a full name, an account number, the starting balance (i.e. *the principal amount of investment*), the period of investment (in years) and annual interest rate (in percentage*)*. You should use **BigDecimal** objects to store the starting balance and the annual interest rate.

The default constructor uses the default constructor of the **Account** class. It initializes the period of investment to 1 year and annual interest rate to 1.25%.

You must code the equals( ), toString( ) and hashCode( ) methods. The toString( ) method uses the toString( ) method of the **Account** class and includes additional information related to the GIC account. It returns a string that has the following format:

```
Name            : Ryan, Mary
Number          : C0005
Current Balance: $1000.00
Account Type                : GIC
Annual Interest Rate        : 1.50%
Period of Investment        : 2 years
Interest Income at Maturity: $30.22
Balance at Maturity         : $1030.22
```

Note: Pay attention to the format (e.g. the %). You need not align the decimal points.

You must override the deposit( ) and withdraw( ) methods of the **Account** class. The deposit( ) method is an empty method and the withdraw( ) method will return false. You must override the *getBalance( )* method such that it returns the final balance of the account (i.e. *balance at maturity*). The balance at maturity is calculated by using the formula in 3.2.a).

You need to implement the *calculateTax( )* method that calculates the amount of tax on the basis of interest income.

3.2. Calculation of Interest Income.

a) **GIC** Account

You should use this online calculator to verify the correctness of your implementation:
https://financialmentor.com/calculator/compound-interest-calculator

(Note: The compounding interval is Annual.)

Here are the formulae for calculating the interest income.

Interest Income = Balance at Maturity – Starting Balance

Balance at Maturity = Current/Starting Balance x (1 + r) ^ t

r = annual interest rate
t = number of years (i.e. period of investment)

(Reminder: You must convert 1.5% into 0.015 for the value of r.)

b) In this assignment, an interest rate is always expressed with two decimal digits (e.g. 3.00, 1.55, 0.15).

3.3 Rules of Taxation

a) The tax rate on any interest income is 15%.

Amount of Tax = Interest income x tax rate.

Notes:
i) You should declare the tax rate as a constant in the **Taxable** interface.
ii) You should use **BigDecmial** objects to do financial calcualtions.

b) For a **GIC** account, the amount of interest income at the end of the investment period is used for tax calculation.

Part 1B: Specification of the Bank Class.

1B.1 API of the Bank Class

```
Bank( )
Bank( String bankName )

String toString( )
int hashCode( )
boolean equals( Object obj )

boolean addAccount( Account account)
Account removeAccount(String accountNumber)
Account [ ] searchByBalance(double balance)

Account [ ] searchByAccountName( String accountName )
Account [ ] getAllAccounts( )
```

1B.2 Functional Requirements

Refer to the functional requirements of the **Bank** class in the lab problems. You need to implement the *searchByAccount( )* and *getAllAccounts( )* methods. As for the *searchByAccountName( )* method, it is similar to that of the searchByBalance( ) method. The difference is that it uses the name of an account (e.g. “Dobson, John”) as the search key. The *getAllAccounts( )* method returns all accounts that have been opened at the bank.

Part 2A: Specification of the FinancialApp Class.

Design and code a standalone Java application class called **FinancialApp**. This application class has the main method that calls other static methods. The main method displays a menu and allows a user to do business with a bank. Here is an example of the menu.

```
Welcome to Seneca@York Bank!
1. Open an account.
2. Close an account.
3. Deposit money.
4. Withdraw money.
5. Display accounts.
6. Display a tax statement.
7. Exit
```

For each menu choice (1-6), the program must display enough information such that the user knows how to enter account information at one line. (Note: Semicolons are used to separate date data and may be followed by zero or more blank spaces.) Here is an example.

```
Please enter your choice>1
Please enter the account type (CHQ/GIC)>GIC
Please enter account information at one line
(e.g. John M. Doe;A1234;1000.00;1.5;2)
>Mary    Ryan;  C0005;1000.00;  1.5;2
```

After the user has entered data values for a given menu choice, the program performs the appropriate operation and displays meaningful result (e.g. account information) on the computer screen. The user interacts with the menu continuously until one has entered 7. The application then displays a message such as “Thank you!” and terminates.

If the user has entered 5, the application should allow the user to select one of the three options:
a) display all accounts with the same account name,
b) display all accounts with the same final balance,
c) display all accounts opened at the bank.

If the user has entered 6, the application should allow the user to display a tax statement about all taxable accounts held by a particular person (i.e. with the same account name). Here is an example:

```
Name: Ryan, Mary
Tax Rate: 15%

[1]
Account Number : D1234
Interest Income: $500.00
Amount of Tax  : $ 75.00
[2]
Account Number : E1234
Interest Income: $600.00
Amount of Tax  : $ 90.00
```

Note: Pay attention to the format (e.g. alignment of the decimal points).

Notes

1. You should design and code as many static methods as needed such that the main method in this application class is highly modular.

2. You must code at least these four static methods in **the FinancialApp** class:

    void loadBank(Bank bank)
    void displayMenu(String bankName)
     int menuChoice( )
    void displayAccount(Account account).

The loadBank( ) method opens two **Chequing** accounts and two **GIC** accounts with the bank. These four accounts are held by two persons whose full names are “John Doe” and “Mary Ryan”. Each person is holding one Chequing account and one GIC account.

The interest rates of GIC accounts are1.50 and 2.50. The periods of GIC investments are 2 years and 4 years. The starting balances of the GIC accounts are $6000.00 and $15,000.00

The displayMenu( ) method displays a menu on the computer screen. The menuChoice( ) method prompts the user to select a menu choice and returns an integer. The displayAccount( ) method displays information about a particular account. (Note: This method will certainly be *polymorphic*.)

Part 2B: Testing Requirements

1. You must test run the main method in the **FinancialApp** class for all the menu choices.

2. If the output of the testing program is correct, save it to a text file (or take as many screenshots as needed).

3. The output must not contain any spelling errors. Otherwise you will lose 21% immediately (B+ max).

Part 3A: Creation of Java Packages.
You must organize your Java classes into different Java packages:

a) package name: **com.seneca.accounts**
   classes: Account, Chequing, GIC

b) package name: **com.seneca.business**
   class: Bank

c) package name: **com.btp400**
   class: FinancialApp.

Otherwise you will lose 21% (B+ max).

Part 3B: Testing and Submission Requirements.
You must submit a JAR file (not zip or rar file) of all source code, *including directory structures and JUnit tests*. (See Part 4 for JUnit tests.) If you use Eclipse to generate the JAR file, make sure that you do the following testing in a command prompt window **before** submitting your work:
- Extract contents from the JAR file.
- Compile and run the Java code.

You must submit a README file that includes command-line instructions for extracting the contents of your JAR file, compiling and running your Java application from a working directory (e.g. c:\testing).

You will lose 21% (B+ max) if your command-line instructions do not allow you or me to compile and run your Java packages successfully. You will lose 31% (C+ max) if the JAR file is empty or does not have the correct subdirectory structures.) ***

Part 4: JUnit Testing.

You must provide **JUnit** testing code for the **Account, Chequing**, **GIC** and **Bank** classes. You must use **"test.btp400.a1"** as the package name. You should create a suite of tests such that one right click on the test suite will trigger automated JUnit testing in Eclipse. You must use javadoc comments to indicate the authorship of the JUnit tests.

Reminder:

If you are working as a team, the implementation of the Java classes (Account, Chequing and GIC) should be shared evenly among the team members. If one team member is the author of a Java class under test (e.g. Account), another team member should be the author of the JUnit tests for that particular class. The authorship of each Java class must be documented clearly.

4.1 Minimum JUnit Testing Requirements.

1. You must test all the constructors used in Account, Chequing, GIC and Bank classes. In particular, you must test if a Java object is properly initialized if negative values and null references are passed as actual parameters.

2. Create JUnit tests for the withdraw( ) and deposit( ) methods.
   a) the **Chequing** class
      i) negative values are passed as actual parameters
      ii) an attempt is made to exceed the maximum number of transactions
      iii) the balance would become negative
      iv) the two methods have updated the current balances correctly
      v) total amount of service charges has been updated correctly

   b) the **GIC** class
      i) The account is not changed by the deposit and withdraw method.
      ii) The withdraw( ) method has returned false.

3. Create JUnit tests to validate that the calculations of final balances are correct.

4. Create at least two JUnit tests to validate that the calculations of the interest incomes (two GIC accounts) are correct.

5. Create JUnit tests to validate that the searchByBalance( ) and searchByAccountName( ) methods are implemented correctly.

Note: You will lose 21% (B+ max) if you do not comply with the minimum JUnit testing requirements.

Part 5: Submission Requirements (INDIVIDUAL).

You must submit four files at Blackboard:

a) a JAR file of source code (including JUnit tests), ***

b) the output file (a text file or screenshots),

c) the README file (command-line instructions), and

d) peer review (on a scale from 0 to 5).

You must mention the names of all the team members when you submit your individual work at Blackboard. Blackboard submission is final. Only one submission is allowed. No email submission will be accepted.

The JAR file should be named as <username>_a1.jar.

The text file should be named as <username>_a1output.txt.

The README file should be named as <username>_README.txt. It should include a list of command-line instructions that you have used to compile and run your Java application.

The peer review should be named as <username>_review.txt. In this text file, you should assign a score to yourself and your team member, on a scale of 0 (very poor contribution) to 5 (excellent contribution).

Note: You will lose 21% (B+ max) if you do not follow all the submission requirements. You will lose 31% (C+ max) if the JAR file is empty or does not have the correct subdirectory structures.) *

***** Reminder**

1. **If you use Eclipse to export your source code and JUnit tests to JAR files, make sure that you have followed the instructions given in Lab #3.**

2. **Before submitting your work, make sure that you can import the JAR files into Eclipse and run the JUnit tests successfully.**

Appendix: Specification of the Account used in the lab problems.

1. An **Account** object holds a person's full name (string), an account number (string) and the current account balance (double). You must code the zero-argument constructor, a constructor that takes three arguments, and the toString() method. You must also code three setters and three getters.

Notes:

a) If a null value is passed as a parameter, the constructor or method must store an empty string ("") in the object.

b) If a negative value is passed as a parameter, the constructor or method must store zero in the object.

b) You should name the setters as setFullName, setAccountNumber and setAccountBalance.

c) You should name the getters as getFullName, getAccountNumber and getAccountBalance.

2. Reuse the **Account** class from Lab Activity #1. Override the **equals( )** method in the **Account** class. Here are the rules of object equality: Two Account objects are equal if they have the same account names, same account numbers and same account balances.

Notes

a) In Lab #1, the data type of the field balance should be double, not int. If not, please make the change now.

b) When you implement the equals( ) method, you should convert the balances into BigDecimal objects. Then you will use the equals( ) method in the BigDecimal class. You should do a little bit of research reading here and find out about the BigDecimal class in the Java API. Here are the links:

https://www.javaworld.com/article/2075315/make-cents-with-bigdecimal.html
https://stackoverflow.com/questions/3413448/double-vs-bigdecimal
https://docs.oracle.com/javase/9/docs/api/java/math/BigDecimal.html.