DUE DATE: MARCH 13, 2019 AT 11:59PM

Instructions:

- You must complete the "**Blanket Honesty Declaration**" checklist on the course website before you can submit any assignment.
- Only submit the **java files**. Do ***not*** submit any other files, unless otherwise instructed.
- To submit the assignment, upload the specified files to the **Assignment 3** folder on the course website.
- Assignments must follow the **programming standards** document published on UMLearn.
- After the due date and time, assignments may be submitted but will be subject to a late penalty. Please see the ROASS document published on UMLearn for the course policy for late submissions.
- If you make multiple submissions, only the **most recent version** will be marked.
- These assignments are your chance to learn the material for the exams. Code your assignments independently. We use software to compare all submitted assignments to each other, and **pursue academic dishonesty vigorously**.
- Your Java programs must compile and run upon download, without requiring any modifications.

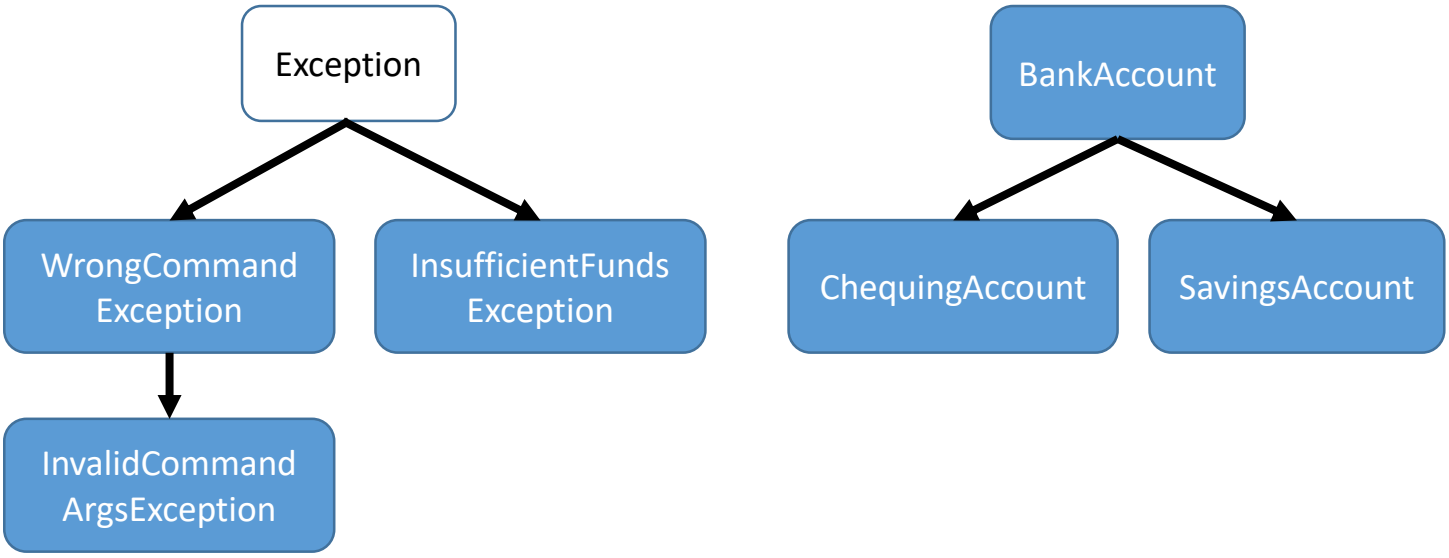
Assignment overview

In this assignment, you have to build a program that reads and processes instructions read from a text file. In other words, the different test phases will be tested using different input files. Your Main class (the class that contains a main method) will handle reading the input files, and will call appropriate methods from the other classes (mainly the Bank class) depending on the instructions given in the input text file.

You will implement a set of related classes of objects:

- **`Main`**: has a main method, defines a constant containing an input filename (more details below), has methods to deal with reading and processing the input file, and sends requests to the Bank class
- **`Bank`**: keeps track of BankClients and BankAccounts, handles all the requests sent from the Main class
- **`BankClient`**: A specification for a client of a Bank

You will also implement these hierarchies of objects (except the Exception class in the white box, because it already exists):



As usual, keep all of your methods short and simple. In a properly-written object-oriented program, the work is distributed among many small methods, which call each other. **The work of the Main class (reading the file, parsing a command/instruction, processing the command) must be divided into different methods (don't just put everything in the main method)**. If you divide (and subdivide) the work correctly, most of these methods will be very short, and the longest ones should not contain more than 30 lines of code (not counting comments, blank lines, or {} lines). Outside of the Main class, all methods should be very short (less than 10 lines).

Unless specified otherwise, all instance variables must be **`private`** (or protected in hierarchies), and all methods should be **`public`** (except helper methods, which can be private). Also unless specified otherwise, there should be ***no*** **println** statements in your classes, except in the Main class. Objects usually do not print anything, they only return **String** values from **toString** methods.

You will notice that this assignment is a little bit more open ended than the previous ones (for example, the parameters that you need to use for the required methods are not always specified, nor are the types for the different instance or class variables). The goal is to get you to think more about how to properly design your program.

Testing Your Coding

We have provided sample ***input*** test files for each phase to help you see if your code is working according to the assignment specifications. These files are *starting* points for testing your code. Part of developing your skills as a programmer is to think through additional important test cases and to write your own code to test these cases.

Phase 1: Building the Main class

First, implement the Main class.

The **`Main`** class should contain:

- A **constant**, containing the filename corresponding to the last phase you completed (i.e. either inputPhase1.txt, inputPhase2.txt, inputPhase3.txt or inputPhase4.txt). You can assume that the input file will be in the same folder as the code when the markers test it.
- A **main** method, which will call the following method (readInputFile) to start reading the file.
- A **readInputFile** method, which gets the filename as a parameter. This method should open the file for reading, read it line by line and call the following method (parseCommand) to process each line.
- A **parseCommand** method, which receives one line of the file (one line in the file = one command to process) and determines which type of command this is. The first token of the line determines the type of the command. Once the command is identified, you should call the specific method that is built to handle that command, like the following one (processComment). Each possible command, once identified, will be handled by a specific method. In phase 1, only the comment commands will be handled, but more types of commands will be added in each phase, so you will be updating the Main class continuously.
- A **processComment** method, which will handle a comment command. Each line in the input text file that starts with the **#** symbol (as the first token) is a comment and should be printed to the console as it appears in the input text file.

You can test your program by running it with the inputPhase1.txt file. You should get the output shown below:

 # This is a test
#
 # This is another test
 # The next line (which is empty) should just be ignored
 # Final test of the comments

Phase 2: Bank and BankClient

Next, implement the **Bank** and the **BankClient** classes.

The **Bank** class will have to do some work for every command that is not a comment. In this phase, it will store all BankClient objects that are created for example. In this phase, the **Bank** class should have:

- As an instance variable: a partially filled array of BankClient objects (and all other instance variables that you might need to handle a partially filled array, as usual)
- A constructor with no parameters that initializes the instances variables.
- A method **addBankClient** that receives a first name and a last name as parameters, assigns a unique id to the BankClient (BankClient ids should have 16 digits, the first four digits must be "1020", and then the last 12 digits will be 000000000000 for the first BankClient and incremented by 1 for each new BankClient), creates the BankClient by calling its constructor and adds it to the partially filled array.
- A method **getBankClient** that receives a first name and a last name as parameters and returns the BankClient in the partially filled array corresponding to those names, or null if the BankClient does not exist.

Then, implement a **BankClient** class, which should have:

- Three instance variables: the first name, the last name and the id of the BankClient (as assigned by the addBankClient method of Bank).
- A constructor that has three parameters, in the above order, which are used to initialize the three instance variables.
- Accessor methods, as needed.
- A **toString** method that returns a String that looks like this: "firstName lastName (id)" (see output below for an example).

Update your Main class so that it now creates 1 Bank object at the start of the main method, which will be used throughout the execution of the program. Then update your Main class to handle the following commands from the input file, following the guidelines explained in phase 1 for dividing and subdividing the work:

- **NEW-CLIENT firstName lastName**
  This adds a new BankClient to the Bank. The second and third tokens represent the first name and the last name of the new BankClient. Once the client has been added to the Bank successfully, print "NEW CLIENT CREATED".
- **GET-CLIENT-INFO firstName lastName**
  This finds the client with the specified first name (2nd token) and last name (3rd token) and prints a description of the client if it exists. If the client does not exist (has not been created previously), "CLIENT NOT FOUND" should be printed instead.

You can test your program by running it with the inputPhase2.txt file. You should get the output shown below:

 # This is a test
#
 # This is another test
 # The next line (which is empty) should just be ignored
 # Final test of the comments
#
 # Testing creating a client
NEW CLIENT CREATED
 # Printing client info
Johnny Cage (1020000000000000)
#
 # Testing creating another client
NEW CLIENT CREATED
 # Printing client info
Sonia Blade (1020000000000001)
#
 # Testing printing client info for a client that does not exist
CLIENT NOT FOUND

Phase 3: Creating new types of Exceptions and handling them

Next implement the **WrongCommandException** class (a subclass of **Exception**), and the **InvalidCommandArgsException** class (a subclass of **WrongCommandException**). Those classes are very simple, since they will only define a constructor receiving a message to be printed as the parameter, and call the super constructor with that message. These new types of exceptions should be thrown and caught in your **Main** class. Make sure that these exceptions do not stop the execution of your program (you need to catch them, and print their String description in the console, without the stack trace). Below is a description of when these exceptions should be thrown.

WrongCommandException:

- This exception is thrown whenever a command (the first token on a line) is not recognized (not one of the commands described here). The message associated with this exception should be:
  "The command [COMMAND] is not recognized." → where [COMMAND] is the first token on the line. See the output below for an example.

InvalidCommandArgsException:

- Whenever the command does not have enough arguments, or the arguments are of the wrong types (e.g. you expect a double and it's a String), throw this exception. The message associated with this exception should be:
  "[COMMAND]: description of which arguments could be missing" → where [COMMAND] is the command that does not have enough arguments, and the description explains what is missing. See the output below for examples.

You can test your program by running it with the inputPhase3.txt file. You should get the output shown below:

```
# This is a test
#
# This is another test
# The next line (which is empty) should just be ignored
# Final test of the comments
#
# Testing creating a client
NEW CLIENT CREATED
# Printing client info
Johnny Cage (1020000000000000)
#
# Testing creating another client
NEW CLIENT CREATED
# Printing client info
Sonia Blade (1020000000000001)
#
# Testing printing client info for a client that does not exist
CLIENT NOT FOUND
#
# Testing wrong commands
# This should throw an InvalidCommandArgsException
InvalidCommandArgsException: GET-CLIENT-INFO: First or last name is missing.
# This should throw a WrongCommandException
WrongCommandException: The command WRONG is not recognized.
# This is another InvalidCommandArgsException
InvalidCommandArgsException: NEW-CLIENT: First or last name is missing
```

Phase 4: BankAccount hierarchy and one new type of Exception

In this phase, you will create the hierarchy of **BankAccount** classes shown on page 1, update the **Bank** class to make use of it, create a new type of exception, and update the **Main** class to handle new commands described below. This is the main part of the assignment. When building these classes and updating previous ones, you need to make full use of the concepts of inheritance, overriding and polymorphism.

First, implement an abstract **BankAccount** class, which should have:

- Three instance variables: an owner (BankClient), a balance (double) and an account id (int).
- A constructor that has three parameters, in the above order, which are used to initialize the three instance variables.
- Accessors as needed.
- A **deposit** method, which receives an amount as a parameter and adds it to the balance.
- A **withdraw** method, which receives an amount as a parameter and subtracts it from the balance, only if there is enough money in the account for the withdrawal. If there is not enough money, the method should throw an **InsufficientFundsException** (see description below).
- A **toString** method that returns a String containing the id, a description of the owner and the balance with 2 digits after the decimal point. This method will be used by the subclasses (see description of the toString methods of **ChequingAccount** and **SavingsAccount**).
- Any additional methods that you might need for polymorphism… more clues below.

Then, implement the **ChequingAccount** class as a subclass of **BankAccount**. It should have:

- A class constant representing a transaction fee of **4.95** that will be applied to withdrawals only.
- A constructor with the same parameters as the super class.
- An overridden **withdraw** method, which will apply the transaction fee after withdrawing the amount given as a parameter. Note that the same rules apply for throwing the **InsufficientFundsException** (the amount to be withdrawn cannot exceed the balance), but the transaction fee can bring the balance to a negative value.
- An overriden **toString** method, making use of the super version, which will return a String consisting of three lines (see output below for an example):
  "Chequing account id: accountId
  Owner: firstName lastName (BankClientId)
  Balance: $xx.xx"

Then, implement the **SavingsAccount** class as a subclass of **BankAccount**. It should have:

- A class constant representing an interest rate of **1.5%**.
- A constructor with the same parameters as the super class.
- A **collectInterest** method that calculates the interest (using the interest rate constant) on the balance and adds it to the balance. You will need to use polymorphism with this method, so make sure there is a version in the base class.
- An overriden **toString** method, making use of the super version, which will return a String consisting of three lines (see output below for an example):
  "Savings account id: accountId
  Owner: firstName lastName (BankClientId)
  Balance: $xx.xx"

Then, implement the **InsufficientFundsException** class as a subclass of **Exception**. It will be thrown by the **withdraw** method when the balance is insufficient, with the associated message: "Insufficient funds for the withdrawal".

Then, update the **Bank** class. It should now have:

- As an instance variable: a partially filled array of **BankAccount** objects (and all other instance variables that you might need to handle a partially filled array, as usual)
- An updated constructor to initialize the new instance variables.

- Methods **addChequingAccount** and **addSavingsAccount** that both receive a **BankClient** and a balance as parameters, assign a unique id to the new account (**BankAccount** ids should have 6 digits, the first digit must be "5", and then the last 5 digits will be 00000 for the first account and increment by 1 for each new account), creates the appropriate subtype of **BankAccount** by calling the correct constructor and adds it to the partially filled array.
- A method **getBankAccount** that receives an account id as a parameter and returns the **BankAccount** in the partially filled array corresponding to this number, or null if the **BankAccount** does not exist.
- A method **payInterest** that iterates through all the existing **BankAccount** objects and pays interest only if applicable (i.e. only **SavingsAccount** receive interest) → you need to use polymorphism here, i.e. call a specific method on all **BankAccount** objects.

Then update your Main class to handle the following new commands from the input file, following the guidelines explained in phase 1 for dividing and subdividing the work:

- **NEW-ACCOUNT TYPE firstName lastName initialBalance**
  This adds a new **BankAccount** of the specified **TYPE** (either **CHQ** for chequing or **SVG** for savings) to the Bank. The third and fourth tokens represent the first name and the last name of the **BankClient** who opens the account (if the client does not exist, print "CLIENT NOT FOUND"). The fifth token is the initial balance, which must be a valid double. Once the account has been added to the Bank successfully, print "NEW ACCOUNT OF TYPE [TYPE] CREATED WITH ID = [ID]" → see output below for an example.
- **GET-ACCOUNT-INFO accountId**
  This finds the account with the specified account id (2nd token) and prints a description of the account if it exists. If the account does not exist (has not been created previously), "ACCOUNT NOT FOUND" should be printed instead.
- **DEPOSIT accountId amount**
  This deposits the amount (3rd token) in the specified account id (2nd token). If completed successfully, print "DEPOSIT COMPLETED". If the account does not exist (has not been created previously), "ACCOUNT NOT FOUND" should be printed instead.
- **WITHDRAW accountId amount**
  This withdraws the amount (3rd token) from the specified account id (2nd token). If completed successfully, print "WITHDRAWAL COMPLETED". If the account does not exist (has not been created previously), "ACCOUNT NOT FOUND" should be printed instead.
- **INTEREST**
  This should trigger the **payInterest** method of the Bank. After completion, "INTEREST WAS PAID" should be printed.

NOTE: you still have to throw **InvalidCommandArgsExceptions** with an appropriate message if these new commands don't have the required arguments. See the output below for examples.

You can test your program by running it with the inputPhase4.txt file. You should get the output shown below:

```
# This is a test
#
# This is another test
# The next line (which is empty) should just be ignored
# Final test of the comments
#
# Testing creating a client
NEW CLIENT CREATED
# Printing client info
Johnny Cage (1020000000000000)
#
# Testing creating another client
NEW CLIENT CREATED
# Printing client info
```

```
Sonia Blade (1020000000000001)
#
# Testing printing client info for a client that does not exist
CLIENT NOT FOUND
#
# Testing wrong commands
# This should throw an InvalidCommandArgsException
InvalidCommandArgsException: GET-CLIENT-INFO: First or last name is missing.
# This should throw a WrongCommandException
WrongCommandException: The command WRONG is not recognized.
# This is another InvalidCommandArgsException
InvalidCommandArgsException: NEW-CLIENT: First or last name is missing
#
# Creating a new chequing account with 0 balance for Johnny Cage
NEW ACCOUNT OF TYPE CHQ CREATED WITH ID = 500000
# Printing account info for previously created account (id = 500000)
Chequing Account id: 500000
Owner: Johnny Cage (1020000000000000)
Balance: $0.00
#
# Testing creating a chequing account for a client that does not exist
CLIENT NOT FOUND
# Testing creating an account of a type that does not exist
InvalidCommandArgsException: NEW-ACCOUNT: IDK is not a valid account type.
# Testing creating an account with an incorrect number of arguments
InvalidCommandArgsException: NEW-ACCOUNT: account type, first name, last name
or balance is missing.
#
# Creating an account for Sonia Blade with 1000 balance
NEW ACCOUNT OF TYPE CHQ CREATED WITH ID = 500001
# Printing account info for previously created account (id = 500001)
Chequing Account id: 500001
Owner: Sonia Blade (1020000000000001)
Balance: $1000.00
#
# Printing account info for unexisting account
ACCOUNT NOT FOUND
# Printing account info without enough arguments
InvalidCommandArgsException:  GET-ACCOUNT-INFO:  accountId  is  missing  or
incorrect.
#
#  Testing  withdrawal  on  chequing  account  with  0  balance:  500000  (throws
InsufficientFundsException)
InsufficientFundsException: Insufficient funds for the withdrawal
# Balance has not changed for this account (still 0)
Chequing Account id: 500000
Owner: Johnny Cage (1020000000000000)
Balance: $0.00
#
# Testing deposit on the chequing account 500000 (balance is now $100.00)
DEPOSIT COMPLETED
# Printing new state of the account 500000
Chequing Account id: 500000
Owner: Johnny Cage (1020000000000000)
Balance: $100.00
#
# Testing withdrawal of 500 from account 500001 (balance was $1000)
```

```
WITHDRAWAL COMPLETED
# Printing new state of the account 500001 (balance should be $500 - transaction
fees = $495.05)
Chequing Account id: 500001
Owner: Sonia Blade (1020000000000001)
Balance: $495.05
# Trying to withdraw another 500 from account 500001 is now impossible
InsufficientFundsException: Insufficient funds for the withdrawal
# But we can withdraw 495 (the fees, but only the fees, can bring the balance
below 0)
WITHDRAWAL COMPLETED
# Printing new state of the account 500001 (balance should be $0.05 - transaction
fees = $-4.90)
Chequing Account id: 500001
Owner: Sonia Blade (1020000000000001)
Balance: $-4.90
#
# Testing DEPOSIT on account that does not exist
ACCOUNT NOT FOUND
# Same for WITHDRAW on unexisting account
ACCOUNT NOT FOUND
# If the amount is not a correct number, throws an InvalidCommandArgsException
InvalidCommandArgsException: DEPOSIT: accountId or amount is missing or
incorrect.
#
# Creating a new savings account for Sonia Blade with a $6000 balance
NEW ACCOUNT OF TYPE SVG CREATED WITH ID = 500002
# Printing account info for this new account (id = 500002)
Savings Account id: 500002
Owner: Sonia Blade (1020000000000001)
Balance: $6000.00
# Withdrawing money from the savings account does not involve transaction fees
WITHDRAWAL COMPLETED
# Printing account info now shows the new balance of $5750.00
Savings Account id: 500002
Owner: Sonia Blade (1020000000000001)
Balance: $5750.00
# Deposit works as usual: depositing 750 in savings account 500002
DEPOSIT COMPLETED
# Printing account info now shows the new balance of $6500.00
Savings Account id: 500002
Owner: Sonia Blade (1020000000000001)
Balance: $6500.00
#
# Creating a new savings account for Johnny Cage with a balance of $10
NEW ACCOUNT OF TYPE SVG CREATED WITH ID = 500003
# Printing account info for this new account (id = 500003)
Savings Account id: 500003
Owner: Johnny Cage (1020000000000000)
Balance: $10.00
#
# Now using the INTEREST command, only the savings account will collect interest
INTEREST WAS PAID
# Now checking the balance of chequing account 500000 (no change to balance --
> still $100.00)
Chequing Account id: 500000
Owner: Johnny Cage (1020000000000000)
```

```
Balance: $100.00
# Now checking the balance of chequing account 500001 (no change to balance --
> still $-4.90)
Chequing Account id: 500001
Owner: Sonia Blade (1020000000000001)
Balance: $-4.90
# Now checking the balance of savings account 500002 (interest was collected -
-> balance is now $6597.50)
Savings Account id: 500002
Owner: Sonia Blade (1020000000000001)
Balance: $6597.50
# Now checking the balance of savings account 500003 (interest was collected -
-> balance is now $10.15)
Savings Account id: 500003
Owner: Johnny Cage (1020000000000000)
Balance: $10.15
```

Hand in

Submit your nine Java files (**`Main.java`**, **`Bank.java`**, **`BankClient.java`**, **`BankAccount.java`**, **`ChequingAccount.java`**, **`SavingsAccount.java`**, **`WrongCommandException.java`**, **`InvalidCommandArgsException.java`**, **`InsufficientFundsException.java`**). ***Do not submit .class or .java~ files!*** You do ***not*** need to submit the **`inputPhaseX.txt`** files that were given to you. **Remember to use the constant in the Main class to specify which input test file corresponds to the last phase that was successfully completed.** If your code fails to compile and run, you will lose ***all*** of the marks for the test runs. The marker will ***not*** try to run anything else, and will ***not*** edit your files in any way. ***Make sure none of your files specify a package at the top!***