Project Requirements

Create a new Eclipse workspace named "**`Project_1234567890`**" on the desktop of your computer (replace **`1234567890`** with your student ID number). For each question below, create a new project in that workspace. Call each project by its question number: "**`Question1`**", "**`Question2`**", etc. If you do not remember how to create a workspace or projects, read the "*Introduction to Eclipse*" document which is on iSpace. Answer all the questions below. At the end of the project, create a ZIP archive of the whole workspace folder. The resulting ZIP file must be called "**`Project_1234567890.zip`**" (replace **`1234567890`** with your student ID number). Upload the ZIP file on iSpace.

Here are a few extra instructions:

- Give meaningful names to your variables so we can easily know what each variable is used for in your program.
- Put comments in your code (in English!) to explain WHAT your code is doing and also to explain HOW your program is doing it. Also put comments in your tests.
- Make sure all your code is properly indented (formatted). Your code should be beautiful to read.

Failure to follow these instructions will result in you losing points.

Question 1

In this project you need to write software for a bank company. The bank has many bank accounts for different customers. There are two types of bank accounts: credit accounts and student accounts.

Write an **`IAccount`** interface for bank accounts, with the following UML specification:

```
+-----------------------------------+
|          <<interface>>            |
|             IAccount              |
+-----------------------------------+
| + getName(): String               |
| + getMoney(): int                 |
| + withdraw(int amount): void      |
+-----------------------------------+
```

and an **`Account`** class that implements **`IAccount`** and has the following UML specification:

```
+------------------------------------+
|               Account              |
+------------------------------------+
| - name: String                     |
| - money: int                       |
+------------------------------------+
| + Account(String name, int money)  |
| + getName(): String                |
| + getMoney(): int                  |
| # setMoney(int money): void        |
| + withdraw(int amount): void       |
| + testAccount(): void              |
+------------------------------------+
```

The **`name`** instance variable indicates the name of the customer for the bank account. The **`money`** instance variable indicates the amount of money which is currently available in the account (to simplify the assignment we will assume that all amounts of money are always integers).

The **`setMoney`** method is **`protected`**, not **`public`**. This means that only subclasses of the **`Account`** class can use the **`setMoney`** method. All the other classes in the software cannot use the **`setMoney`** method, so they cannot change the amount of money in an account. This is good for the security of the banking software you are writing!

The purpose of the **`withdraw`** method is to withdraw (subtract) the amount of money given as argument to the method from the amount of money currently stored in the account. The **`withdraw`** method of the **`Account`** class is **`abstract`** since different types of bank accounts will withdraw money in different ways.

Also add to your program a **`Test`** class to test your **`Account`** class.

Question 2

Add a class **`CreditAccount`** that extends **`Account`**. The constructor of the **`CreditAccount`** class takes a name and an amount of money as arguments. The **`CreditAccount`** class does not have any instance variable.

The **`withdraw`** method of the **`CreditAccount`** class simply subtracts the amount of money given as argument to the method from the amount of money currently stored in the account. A credit account is allowed to have a negative amount of money in the account, so money can be withdrawn from a credit account even if this makes the amount of money in the credit account become negative.

Make sure you test all the methods of your new **`CreditAccount`** class, including inherited methods.

Question 3

Add a class **`StudentAccount`** that extends **`Account`**. The constructor of the **`StudentAccount`** class takes a name and an amount of money as arguments. If the amount of money given as argument is strictly less than zero then the constructor must throw a **`NotEnoughMoneyException`** with the message **`"Cannot create student account with negative amount of money"`**. The **`StudentAccount`** class does not have any instance variable.

The **`withdraw`** method of the **`StudentAccount`** class subtracts the amount of money given as argument to the method from the amount of money currently stored in the account. A student account is not allowed to have a negative amount of money in the account, so money can be withdrawn from a student account only if the amount of money in the student account will remain positive. If the amount to withdraw is too big then the amount of money in the student account must not change and the **`withdraw`** method must throw a **`NotEnoughMoneyException`** with the message **`"Cannot withdraw XXX yuan from account, only YYY yuan is available"`**, where **`XXX`** is replaced by the amount of money that was given as argument to the withdraw method and **`YYY`** is replaced by the amount of money currently in the student account. For example, if a student currently has 500 yuans in a student bank account and **`withdraw(1000)`** is called then the student still has 500 yuans in the account and the method throws a **`NotEnoughMoneyException`** with the message **`" Cannot withdraw 1000 yuan from account, only 500 yuan is available"`**

Note: to simplify the project, do not worry about the **`setMoney`** method.

Change other classes and interfaces as necessary.

Make sure you test your new **`StudentAccount`** class.

Question 4

Add a **`Bank`** class with the following UML specification:

```
+------------------------------------------+
|                   Bank                   |
+------------------------------------------+
| - name: String                           |
| - accounts: ArrayList<IAccount>          |
+------------------------------------------+
| + Bank(String name)                      |
| + addAccount(IAccount account): void     |
| + totalMoney(): int                      |
| + getMoney(String name): int             |
| + withdraw(String name, int amount): void |
| + testBank(): void                       |
+------------------------------------------+
```

When a bank is created, it has an arraylist of accounts but the arraylist is empty (the arraylist does not contain any bank account).

The **addAccount** method takes an account as argument and adds the account to the arraylist of accounts for the bank.

The **totalMoney** method returns as result the total amount of money in all the bank accounts of the bank.

The **getMoney** method takes as argument the name of a customer and returns as result the amount of money currently stored in the bank account that belongs to that customer. If the customer does not have a bank account in the bank then the **getMoney** method must throw an **UnknownCustomerException** with the message **"Customer XXX unknown"**, where **XXX** is replaced with the name of the customer. Do not worry about multiple customers having the same name.

The **withdraw** method takes as argument the name of a customer and an amount of money and withdraws that amount of money from the amount of money currently stored in the bank account that belongs to that customer. If the customer does not have a bank account in the bank then the **withdraw** method must throw an **UnknownCustomerException** with the message **"Customer XXX unknown"**, where **XXX** is replaced with the name of the customer. Do not worry about multiple customers having the same name.

Note: the **withdraw** method does not catch any exception, it only throws exceptions.

Hint: use the **equals** method to compare strings, not the **==** operator which only works with constant strings.

Make sure you test your new **Bank** class.

Question 5

In this question and the next one we want to create a command line interface (CLI) for our banking software.

Add a **CLI** class with a **main** method. Your code then has two classes with a **main** method: the **Test** class that you can use to run all your tests for all your classes, and the **CLI** class that you will now use to run the interactive text-based interface of your program.

The **CLI** class does not have any **testCLI** method because this class is only used to allow users to use the software interactively.

Add to the **CLI** class a private static **input** instance variable which is a **Scanner** object that reads input from the standard input stream **System.in**:

```
private static Scanner input = new Scanner(System.in);
```

Always use this **input** scanner object when you need to read input. (Never close this scanner object, because this would also close the standard input stream **System.in**, and then the next time you tried to read something from the standard input stream you would get a **NoSuchElementException**!)

In addition to the **main** method and the **input** instance variable, the **CLI** class has two methods called **readLine** and **readPosInt**.

The **readLine** method is static and private, it takes a string as argument, and returns another string as result. The **readPosInt** method is static and private, it takes a string as argument, and returns a positive integer as result.

The **readLine** method uses **System.out.print** (not **println**) to print its string argument on the screen (later when we use the **readLine** method, the string argument of the method will be a message telling the user to type some text). Then the **readLine** method uses the **input** scanner object to read a whole line of text from the user of the program and returns the text as result.

The **readPosInt** method uses **System.out.print** (not **println**) to print its string argument on the screen (later when we use the readPosInt method, the string argument of the method will be a message telling the user to type some integer). Then the **readPosInt** method uses the **input** scanner object to read an integer from the user of the program.

After reading the integer, the **readPosInt** method must also use the scanner's **nextLine** method to read the single newline character that comes from the user pressing the **Enter** key on the keyboard after typing the integer (if you do not read this newline character using the **nextLine** method inside the **readPosInt** method, then the newline character will remain in the input stream, and, the next time you use the **readLine** method described above, the **readLine** method will just immediately read only the newline character from the input stream and return an empty string as result, without waiting for the user to type anything!)

If the user types something which is not an integer, then the **nextInt** method of the scanner will throw an **InputMismatchException**. In that case the code of your **readPosInt** method must catch the exception, use **System.out.println** to print the error message **"You must type an integer!"** to the user (use **System.out.println** for this, not **System.err.println**, otherwise you might hit a bug in Eclipse...), use the scanner's **nextLine** method to read (and ignore) the wrong input typed by the user of the program (if you do not do this, the wrong input typed by the user will remain in the input stream, and the next time you call the **nextInt** method again, you will get an **InputMismatchException** again!), and then do the whole thing again (including printing again the string argument of the **readPosInt** method) to try to read an integer again (hint: put the whole code of the method inside a **while** loop).

After reading the integer and the newline character (which is just ignored), the **readPosInt** method tests the integer. If the integer is bigger than or equal to zero, then the **readPosInt** method returns the integer as result. If the integer is strictly less than zero, then the **readPosInt** method uses **System.out.println** to print the error message **"Positive integers only!"** to the user (use **System.out.println** for this, not **System.err.println**, otherwise you might hit a bug in Eclipse...), and then does the whole thing again (including printing again the string argument of the **readPosInt** method) to try to read an integer again (hint: just print the error message, and then the **while** loop you already have around the whole code will automatically do the whole thing again...)

For example, if you want to check that your two methods **readLine** and **readPosInt** work correctly, put the following code in the **main** method of your **CLI** class:

```java
public static void main(String[] args) {
      String str1 = readLine("Type some text: ");
      System.out.println("Text read is: " + str1);
      int i = readPosInt("Type an integer: ");
```

```
        System.out.println("Integer read is: " + i);
        String str2 = readLine("Type some text again: ");
        System.out.println("Text read is: " + str2);
    }
```

then running the **main** method of the **CLI** class should look like this (where `aaaa bbbb`, `cccc`, `dddd eeee`, `-100`, `-200`, `1234`, and `ffff gggg` are inputs typed by the user on the keyboard):

```
    Type some text: aaaa bbbb
    Text read is: aaaa bbbb
    Type an integer: cccc
    You must type an integer!
    Type an integer: dddd eeee
    You must type an integer!
    Type an integer: -100
    Positive integers only!
    Type an integer: -200
    Positive integers only!
    Type an integer: 1234
    Integer read is: 1234
    Type some text again: ffff gggg
    Text read is: ffff gggg
```

Question 6

Once you have checked that your methods **readLine** and **readPosInt** work correctly, remove all the code inside the **main** method of the **CLI** class so that the **main** method is empty again.

In the rest of this question, use the **readLine** and **readPosInt** methods every time your program needs to read a string or an integer from the user.

In the empty **main** method of the **CLI** class, create a single **Bank** object with the name **"UIC Bank"**. The **main** method of the **CLI** class must then print a menu that allows the user of your software to do six different actions that involve the bank object, and your program must then read an integer from the user that indicates which action must be performed by the program (see below for the details about each action). Use the **readPosInt** method to print the menu (give the string for the menu as the argument of **readPosInt**) and to read the integer typed by the user.

For example, the menu should look like this:

```
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6):
```

The user then types an integer between 1 and 6 to select the action.

For example (where 3 is an input from the user):

```
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 3
```

and your program then performs the selected action.

After an action has been performed by your program, your program must again print the menu and ask again the user of the program for the next action to perform (hint: put the whole code of the **main** method inside a **while** loop, except for the one line of code that creates the single bank object).

If the user types an integer which is not between 1 and 6, then your program must print an error message **"Unknown action!"** to the user (hint: when testing the integer for the action, use the **default** case of a **switch** statement) and then print the menu again (by just going back to the beginning of the **while** loop).

For example (where 7 is an input from the user):

```
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 7
Unknown action!
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6):
```

If the user types something which is not an integer, the **`readPosInt`** method that you implemented in the previous question will automatically repeat the menu and ask the user to type an integer again until the user actually types an integer, so you do not have to worry about this in the code of the **`main`** method of your **`CLI`** class.

For example (where `aaaa` and `bbbb` are inputs from the user):

```
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): aaaa
You must type an integer!
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): bbbb
You must type an integer!
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6):
```

Here are the detailed explanations for each action.

Action 1: printing the total amount of money stored in the bank.

When the user of the software specifies action 1, your program must simply print on the screen the total amount of money currently stored in the bank object for "UIC Bank". Then your program goes back to printing the menu of actions (by just going back to the beginning of the **`while`** loop).

For example (where `1` is an input from the user):

```
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 1
Total amount of money in the bank: 2000
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6):
```

Action 2: adding a new customer account to the bank.

When the user of the software specifies action 2, your program must add a new account to the bank. To add a new account, your program needs to ask the user three things: the type of account (an integer read using **`readPosInt`**: the integer **`1`** represents a credit account, the integer **`2`** represents a student account, any other integer must result in an error message **`"Unknown type of account!"`** being printed and the software going immediately back to the main menu), the name of the customer (a string read using **`readLine`**), and the initial amount of money that the customer puts into the account when the account is created. You program must then create the correct account, add it to the bank, and print an information message for the user. The program then goes back to the menu.

For example (where `2`, `3`, `2`, `1`, `Philippe`, `500`, `2`, `2`, `Meunier`, and `1500` are inputs from the user):

```
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 2
Type the account type (credit:1 student:2): 3
Unknown type of account!
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 2
Type the account type (credit:1 student:2): 1
Enter the name of the customer: Philippe
Enter the initial amount of money: 500
Credit account for Philippe with 500 yuan has been added
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 2
Type the account type (credit:1 student:2): 2
Enter the name of the customer: Meunier
Enter the initial amount of money: 1500
Student account for Meunier with 1500 yuan has been added
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6):
```

Note that the **`readPosInt`** method prevents the initial amount of money from being negative, so the constructor for the **`StudentAccount`** class will never throw a **`NotEnoughMoneyException`** when you create a student account object. Nevertheless the code of the **`main`** method of your **`CLI`** class must handle this exception by printing the error message "**BUG! This must never happen!**" and immediately terminating the program using **System.exit(1);**

Action 3: listing the amount of money in the account of a given customer.

When the user of the software specifies action 3, your program must ask the user to type the name of a customer, and the program then prints the amount of money which is currently in the account of this user.

For example (where 3, Philippe, 3, and Meunier are inputs from the user):

```
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 3
Enter the name of the customer: Philippe
Philippe has 500 yuan in the bank
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 3
Enter the name of the customer: Meunier
Meunier has 1500 yuan in the bank
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6):
```

If the name of the customer is wrong (the bank does not have an account for this customer) then an **UnknownCustomerException** exception will be thrown by the **Bank** object. The code of the **main** method of your **CLI** class must catch this exception, print the error message from the exception object, and then it just goes back to printing the menu of actions (by just going back to the beginning of the **while** loop).

For example (where 3 and aaaa are inputs from the user):

```
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 3
Enter the name of the customer: aaaa
Customer aaaa unknown
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6):
```

Action 4: withdrawing money from the account of a given customer.

When the user of the software specifies action 4, your program must ask the user to type the name of a customer, and an amount of money to withdraw, and the program then withdraws that amount of money from that customer's bank account. Then the program goes back to the main menu.

For example (where 3, Philippe, 4, Philippe, 200, 3, and Philippe are inputs from the user):

```
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 3
Enter the name of the customer: Philippe
Philippe has 500 yuan in the bank
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 4
Enter the name of the customer: Philippe
Enter the amount of money to withdraw: 200
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 3
Enter the name of the customer: Philippe
Philippe has 300 yuan in the bank
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6):
```

If the name of the customer is wrong (the bank does not have an account for this customer) then an **UnknownCustomerException** exception will be thrown by the **Bank** object. The code of the **main** method of your **CLI** class must catch this exception, print the error message from the exception object, and then it just goes back to printing the menu of actions (by just going back to the beginning of the **while** loop).

For example (where 4, aaaa, and 200 are inputs from the user):

```
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 4
Enter the name of the customer: aaaa
Enter the amount of money to withdraw: 200
Customer aaaa unknown
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6):
```

If the account of the customer is a student account and the amount of money to withdraw is too big then a **NotEnoughMoneyException** exception will be thrown by the **StudentAccount** object. The code of the **main** method of your **CLI** class must catch this exception, print the error message from the exception object, and then it just goes back to printing the menu of actions (by just going back to the beginning of the **while** loop).

For example (where 3, **Meunier**, 4, **Meunier**, **2000** , 3, and **Meunier** are inputs from the user):

```
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 3
Enter the name of the customer: Meunier
Meunier has 1500 yuan in the bank
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 4
Enter the name of the customer: Meunier
Enter the amount of money to withdraw: 2000
Cannot withdraw 2000 yuan from account, only 1500 yuan is available
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 3
Enter the name of the customer: Meunier
Meunier has 1500 yuan in the bank
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6):
```

Action 5: depositing money into the account of a given customer.

When the user of the software specifies action 5, your program must ask the user to type the name of a customer, and an amount of money to deposit, and the program then deposits that amount of money into that customer's bank account. Then the program goes back to the main menu.

Note: the bank object that you are using does not have a deposit method. So, in the code of the **main** method of the **CLI** class, simulate a deposit by simply doing a withdrawal of the negative amount! For example, depositing 500 yuan on an account is the same as withdrawing -500 yuan from the same account.

For example (where 3, **Philippe**, 5, **Philippe**, **200**, 3, and **Philippe** are inputs from the user):

```
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 3
Enter the name of the customer: Philippe
Philippe has 300 yuan in the bank
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 5
Enter the name of the customer: Philippe
Enter the amount of money to deposit: 200
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 3
Enter the name of the customer: Philippe
Philippe has 500 yuan in the bank
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6):
```

If the name of the customer is wrong (the bank does not have an account for this customer) then an **UnknownCustomerException** exception will be thrown by the **Bank** object. The code of the **main** method of your **CLI** class must catch this exception, print the error message from the exception object, and then it just goes back to printing the menu of actions (by just going back to the beginning of the **while** loop).

For example (where 5, **aaaa**, and **200** are inputs from the user):

```
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 5
Enter the name of the customer: aaaa
Enter the amount of money to deposit: 200
Customer aaaa unknown
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6):
```

Note that, even if a customer has a student account, the **readPosInt** methods prevents the amount to deposit from being negative. So doing a deposit always increases the amount of money stored in an account (or the amount of money remains the same if the customer deposits zero yuan). This means the student account will never throw a **NotEnoughMoneyException**. Nevertheless the code of the **main** method of your **CLI** class must handle this exception by printing the error message "**`BUG! This must never happen!`**" and immediately terminating the program using **`System.exit(1);`**

Action 6: quitting the program.

When the user of the software specifies action 6, your program must print a "**`Goodbye!`**" message, and terminate the program using: **`System.exit(0)`**.

For example (where 6 is an input from the user):

```
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 6
Goodbye!
```

Here is a more complete example of running the software:

```
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): aaaa
You must type an integer!
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): bbbb cccc
You must type an integer!
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): -100
Positive integers only!
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 7
Unknown action!
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 1
Total amount of money in the bank: 0
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 2
Type the account type (credit:1 student:2): -100
Positive integers only!
Type the account type (credit:1 student:2): 3
Unknown type of account!
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 2
Type the account type (credit:1 student:2): 1
Enter the name of the customer: Philippe
Enter the initial amount of money: -100
Positive integers only!
Enter the initial amount of money: 500
Credit account for Philippe with 500 yuan has been added
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 2
Type the account type (credit:1 student:2): 2
Enter the name of the customer: Meunier
Enter the initial amount of money: 1500
Student account for Meunier with 1500 yuan has been added
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 1
Total amount of money in the bank: 2000
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 3
Enter the name of the customer: Philippe
Philippe has 500 yuan in the bank
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 3
Enter the name of the customer: Meunier
Meunier has 1500 yuan in the bank
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 3
Enter the name of the customer: aaaa
Customer aaaa unknown
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 4
Enter the name of the customer: Philippe
Enter the amount of money to withdraw: -100
Positive integers only!
Enter the amount of money to withdraw: 200
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 3
Enter the name of the customer: Philippe
Philippe has 300 yuan in the bank
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 4
Enter the name of the customer: aaaa
Enter the amount of money to withdraw: 200
Customer aaaa unknown
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 4
```

```
Enter the name of the customer: Meunier
Enter the amount of money to withdraw: 2000
Cannot withdraw 2000 yuan from account, only 1500 yuan is available
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 3
Enter the name of the customer: Meunier
Meunier has 1500 yuan in the bank
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 5
Enter the name of the customer: Philippe
Enter the amount of money to deposit: -100
Positive integers only!
Enter the amount of money to deposit: 200
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 3
Enter the name of the customer: Philippe
Philippe has 500 yuan in the bank
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 5
Enter the name of the customer: aaaa
Enter the amount of money to deposit: 200
Customer aaaa unknown
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 1
Total amount of money in the bank: 2000
Type an action (total:1 add:2 list:3 withdraw:4 deposit:5 quit:6): 6
Goodbye!
```

Question 7

We now want to create a graphical user interface (GUI) for our bank software. Since we want our bank software to have multiple views, we will use the Model-View-Controller design pattern.

First, create a **ModelListener** interface with the following UML specification:

```
      +--------------------+
      |   <<interface>>    |
      |   ModelListener    |
      +--------------------+
      | + update(): void   |
      +--------------------+
```

This interface will be implemented by views and the model will use this interface to notify the views that they need to update themselves.

Second, the **Bank** class is the class that contains all the data for the bank. Therefore the **Bank** class plays the role of the model. Therefore the **Bank** class needs to keep an arraylist of model listeners that need to be notified every time the bank (the model) changes. Therefore add to the **Bank** class an arraylist of **ModelListener**. Also add to the **Bank** class an **addListener** method that takes a **ModelListener** as argument and adds it to the arraylist of listeners. Also add to the **Bank** class a private **notifyListeners** method that takes nothing as argument and calls the **update** method of all the listeners of the bank. Then change the **addAccount** and **withdraw** methods so that they call the **notifyListeners** every time a change is made to the bank's data (only the **addAccount** and **withdraw** methods change the bank data, so only these two methods need to call the **notifyListeners** method; the **totalMoney** and **getMoney** methods do not change the bank's data, they only inspect the data, so they do not need to call the **notifyListeners** method).

Use the **Test** class to make sure all your tests still work. Use the **CLI** class to make sure your command line interface still works.

Third, create a **ViewSimple** class that extends **JFrame**, implements the **ModelListener** interface, and has the following UML specification:

```
      +-----------------------------------------------+
      |                ViewSimple                     |
      +------------------------------------------+
      | - m: Bank                                |
      | - c: ControllerSimple                    |
      | - label: JLabel                          |
      +------------------------------------------+
      | + ViewSimple(Bank m, ControllerSimple c) |
      | + update(): void                         |
      +------------------------------------------+
```

The constructor of the **ViewSimple** class registers the view with the model (the bank) using the **addListener** method of the model, creates a **JLabel** object, stores it in the **label** instance variable of the **ViewSimple** class, initializes it to display the total amount of money in all the bank accounts of the bank, and adds the label to the view (which is a frame). The **update** method of the **ViewSimple** class updates the text of the **label** as necessary so that the **label** always displays the current value of the total amount of money in all the bank accounts of the bank.

Fourth, create a **ControllerSimple** class with the following UML specification:

```
+------------------------------------------+
|             ControllerSimple             |
+------------------------------------------+
| - m: Bank                                |
+------------------------------------------+
| + ControllerSimple(Bank m)               |
+------------------------------------------+
```

Since the **ViewSimple** does not have any button, it cannot perform any action, therefore the corresponding controller **ControllerSimple** does nothing. (We still want to have the **ControllerSimple** class so that our application follows the correct Model-View-Controller design pattern.)

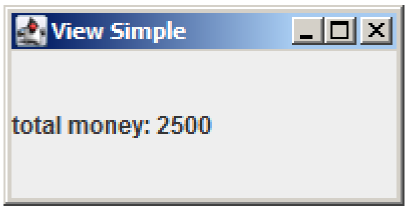
Fifth, create a **GUI** class with a **main** method. In this **main** method, create an anonymous class that implements the **Runnable** interface with a **run** method and use the **javax.swing.SwingUtilities.invokeLater** method to run that code on the event dispatch thread.

Sixth, we need to connect the model, the view, and the controller to each other. So in the **run** method of the anonymous class:

- create a **Bank** object (the model object) with the name **"UIC Bank"**;
- then create a **ControllerSimple** object (the controller object) that takes the model object as argument;
- then create a **ViewSimple** object that takes the model object and the controller object as argument;

Use the **GUI** class to run your GUI: you should see a window that shows the total amount of money in all the bank accounts of the bank. This total amount must be zero, since the bank (model object) you just created above does not contain any account!

As a test, in the **run** method of the anonymous class, you can try to manually add to your bank (model object) some student accounts and credit accounts to check that your GUI displays the correct total amount of money in all the bank accounts of the bank. For example:



total money: 2500

Question 8

In the next questions we want to add more views. So, to simplify the next questions, create a **`View`** class which is going to be the superclass of all views. This **`View`** class is generic, extends **`JFrame`**, implements the **`ModelListener`** interface, and has the following UML specification:

```
+-------------------------------------------+
|        View<T extends Controller>         |
+-------------------------------------------+
| # m: Bank                                 |
| # c: T                                    |
+-------------------------------------------+
| + View(Bank m, T c)                       |
| + update(): void                          |
+-------------------------------------------+
```

The **`m`** and **`c`** instance variables of the **`View`** class are **`protected`** (so that they can be easily used in all the subclasses of **`View`**). In the constructor of the **`View`** class, the view registers itself with the model. The **`update`** method of the **`View`** class is **`abstract`**.

Then modify the **`ViewSimple`** class to be a subclass of the **`View<…>`** class. The **`ViewSimple`** class must then have only one instance variable: the **`label`**. To simplify a little the code of the next questions, also move the **`setDefaultCloseOperation`** method call from the constructor of **`ViewSimple`** to the constructor of **`View`**. Also make sure that the **`ViewSimple`** does not directly register itself with the model anymore, since this is now done in the superclass **`View`**.

Also create a **`Controller`** class which is going to be the superclass of all controllers. This **`Controller`** class has the following UML specification:

```
+-------------------------------------------+
|                Controller                 |
+-------------------------------------------+
| # m: Bank                                 |
+-------------------------------------------+
| + Controller(Bank m)                      |
+-------------------------------------------+
```

The **`m`** instance variable of the **`Controller`** class is protected (so that it can be easily used in all the subclasses of **`Controller`**).

Then modify the **`ControllerSimple`** class to be a subclass of the **`Controller`** class. (Note: since **`ControllerSimple`** does nothing anyway, we could just remove it and replace it with **`Controller`** in the definition of **`ViewSimple`** and in the **`run`** method of the **`GUI`** class, but here we keep **`ControllerSimple`** just to make the Model-View-Controller design pattern very clear.)

Run your GUI and check that it still works as before.

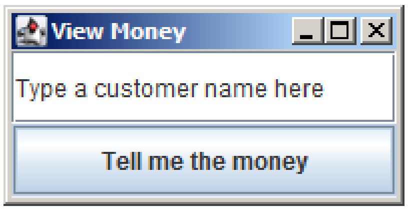
Question 9

We now want to add a new "get money" view that allows the user of the software to check how much money there is in a specific account.

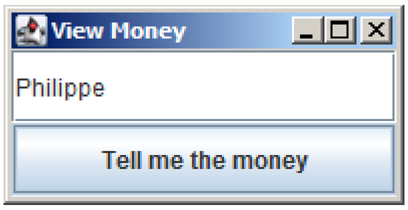
Create a **`ViewGetMoney`** class that extends **`View<ControllerGetMoney>`** and has the following UML specification:

```
+---------------------------------------------+
|                 ViewGetMoney                |
+-----------------------------------------------+
| - t: JTextField                               |
+-----------------------------------------------+
| + ViewGetMoney(Bank m, ControllerGetMoney c)  |
| + update(): void                              |
+-----------------------------------------------+
```

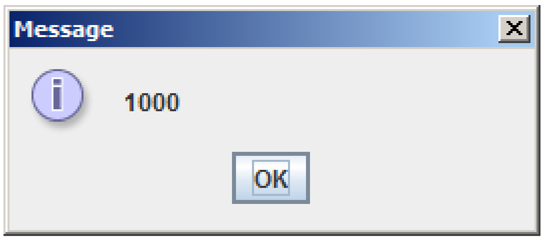
The **ViewGetMoney** shows the text field called **t** (where the user can type text) and a button. Use a grid layout manager to position the two components. For example:



The user can type in the text field **t** the name of a bank account customer. For example:



When the user then clicks on the button, the action listener of the button must read the name of the customer that was typed in the text field (using the **getText** method of the text field) and must call the **getMoney** method of the controller with that customer name as argument. The **getMoney** method of the controller returns a string as result which must then be displayed back to the user using a message dialog (using the **showMessageDialog** method of the **JOptionPane** class). For example:



The **update** method of the **ViewGetMoney** class does nothing, because the **ViewGetMoney** class does not graphically display any data from the bank (the model).

Also create a **ControllerGetMoney** class that extends **Controller** and has the following UML specification:

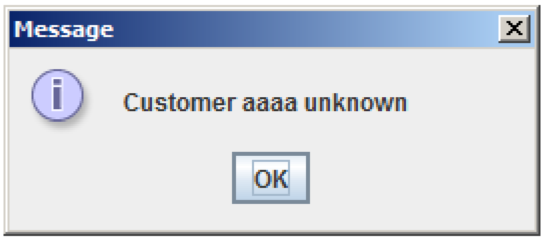
```
+-------------------------------------------+
|            ControllerGetMoney             |
+-------------------------------------------+
+-------------------------------------------+
| + ControllerGetMoney(Bank m)              |
| + getMoney(String name): String           |
+-------------------------------------------+
```

The **getMoney** method takes the name of a bank customer as argument. The **getMoney** method of the controller then calls the **getMoney** method of the bank to get the amount of money currently stored in the bank account that belongs to that customer. The **getMoney** method of the controller then transforms the integer result of the **getMoney** method of the bank into a string and returns that string as result (to the view). If the **getMoney** method of the bank throws an **UnknownCustomerException** then the **getMoney** method of the controller must catch this exception and return as result the error message from the exception object.

Modify the **run** method of the **GUI** class to add a **ViewGetMoney** view that uses a **ControllerGetMoney** controller and the same model as before (not a new model!) Do not delete the previous view.

Run your GUI and check that you can correctly use the new view to query the amount of money for different customers of your bank (obviously your bank must have some customers in it to test this: see the last paragraph of Question 7).

Also check that querying the amount of money of an unknown customer correctly shows an error message. For example:


Message

Customer aaaa unknown

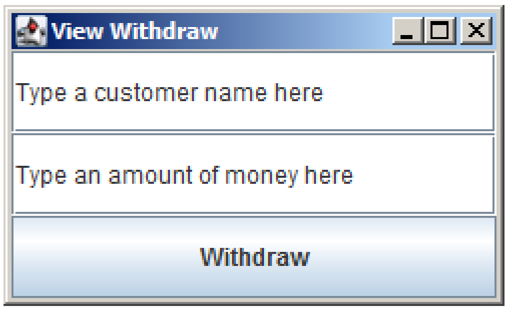
Question 10

We now want to add a new "withdraw" view that allows the user of the software to withdraw money from a specific account.

Create a **ViewWithdraw** class that extends **View<ControllerWithdraw>** and has the following UML specification:

```
+-----------------------------------------------+
|                  ViewWithdraw                 |
+-----------------------------------------------+
| - t1: JTextField                              |
| - t2: JTextField                              |
+-----------------------------------------------+
| + ViewWithdraw(Bank m, ControllerWithdraw c)  |
| + update(): void                              |
+-----------------------------------------------+
```

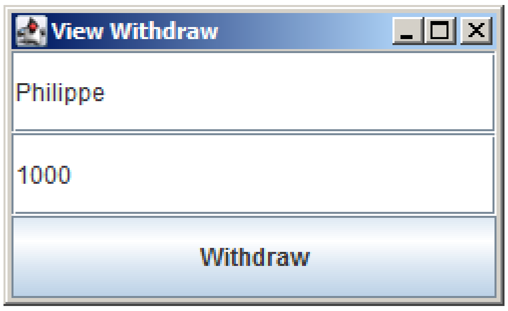
The **ViewWithdraw** shows the two text field called **t1** and **t2** (where the user can type text) and a button. Use a grid layout manager to position the three components. For example:


View Withdraw

Type a customer name here

Type an amount of money here

The user can type in the first text field the name of a bank account customer and can type in the second text field an amount of money. For example:



When the user then clicks on the button, the action listener of the button must read the name of the customer that was typed in the first text field (using the **`getText`** method of the text field) and the amount of money that was typed in the second text field (using again the **`getText`** method) and must call the **`withdraw`** method of the controller with these two strings as arguments. The **`withdraw`** method of the controller then returns a string as result. If the string returned by the **`withdraw`** method of the controller is different from the empty string `""` then this string must be displayed back to the user using a message dialog (using the **`showMessageDialog`** method of the **`JOptionPane`** class). If the string returned by the **`withdraw`** method of the controller is equal to the empty string `""` then nothing happens in **`ViewWithdraw`**.

The **`update`** method of the **`ViewWithdraw`** class does nothing, because the **`ViewWithdraw`** class does not graphically display any data from the bank (the model).

Also create a **`ControllerWithdraw`** class that extends **`Controller`** and has the following UML specification:

```
+-------------------------------------------------+
|                ControllerWithdraw               |
+-------------------------------------------------+
+-------------------------------------------------+
| + ControllerWithdraw(Bank m)                    |
| + withdraw(String name, String amount): String |
+-------------------------------------------------+
```

The **`withdraw`** method takes the name of a bank customer and an amount of money (as a string) as arguments. The **`withdraw`** method of the controller then transforms the amount of money from a string to an integer (using the **`Integer.parseInt`** static method) and calls the **`withdraw`** method of the bank to withdraw that amount of money from the amount of money currently stored in the bank account that belongs to the customer with the correct name.

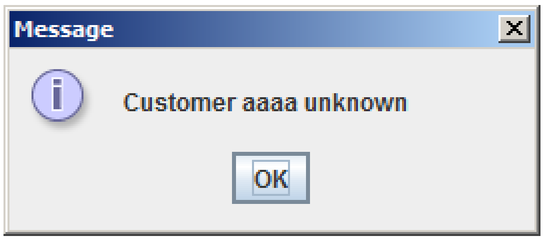
- If no exception occurs then the **`withdraw`** method of the controller returns the empty string.
- If the **`withdraw`** method of the bank throws an **`UnknownCustomerException`** then the **`withdraw`** method of the controller must catch this exception and return as result the error message from the exception object.
- If the **`withdraw`** method of the bank throws a **`NotEnoughMoneyException`** then the **`withdraw`** method of the controller must catch this exception and return as result the error message from the exception object.
- If the **`parseInt`** method of the **`Integer`** class throws a **`NumberFormatException`** (because the user typed something which is not an integer) then the **`withdraw`** method of the controller must catch this exception and return as result the error message from the exception object.

Note: to keep things simple, it is allowed for a user of your software to withdraw a negative amount of money in this “withdraw” view, so there is no need to check for that in the controller.

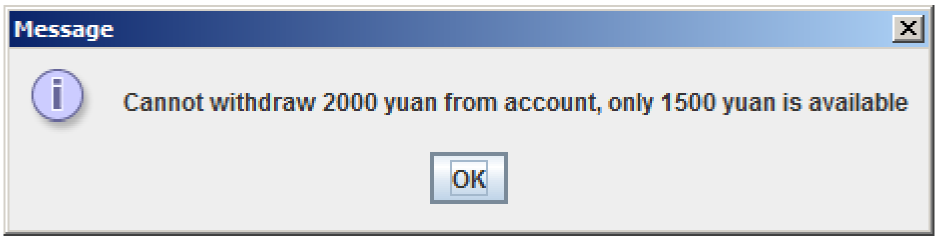
Modify the **`run`** method of the **`GUI`** class to add a **`ViewWithdraw`** view that uses a **`ControllerWithdraw`** controller and the same model as before (not a new model!) Do not delete the previous views.

Run your GUI and check that you can correctly use the new view to withdraw money for different customers of your bank (obviously your bank must have some customers in it to test this: see the last paragraph of Question 7).

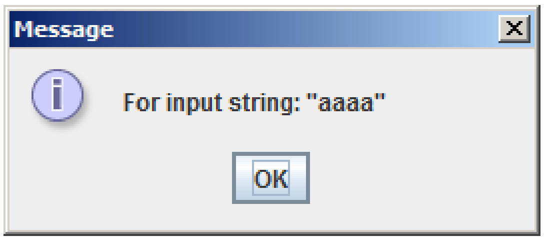
- Check that, when you withdraw money, the simple view is automatically correctly updated to show the new total amount of money in all the bank accounts of the bank.
- Also use the "get money" view to check that the customer's money correctly changed.
- Also check that withdrawing money from an unknown customer correctly shows an error message. For example:


Customer aaaa unknown

- Also check that withdrawing too much money from a student account correctly shows an error message. For example:


Cannot withdraw 2000 yuan from account, only 1500 yuan is available

- Also check that trying to withdraw an amount of money which is not an integer correctly shows an error message (do not worry about the content of the error message). For example:


For input string: "aaaa"

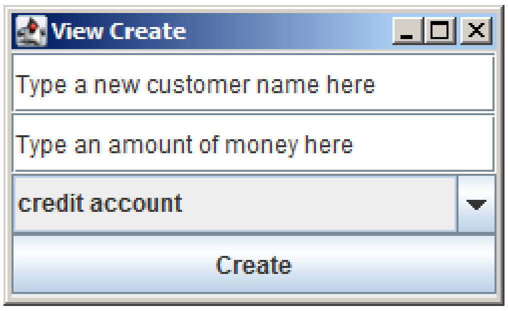
Question 11

We now want to add a new "create" view that allows the user of the software to create a new bank account.

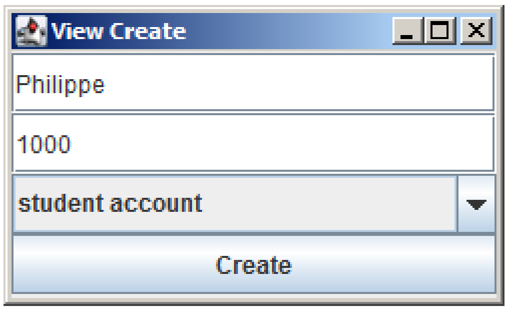
Create a **`ViewCreate`** class that extends **`View<ControllerCreate>`** and has the following UML specification:

```
+-----------------------------------------------+
|                  ViewCreate                   |
+-----------------------------------------------+
| - t1: JTextField                              |
| - t2: JTextField                              |
| - cb: JComboBox<String>                       |
+-----------------------------------------------+
| + ViewCreate(Bank m, ControllerCreate c)      |
| + update(): void                              |
+-----------------------------------------------+
```

The **`ViewCreate`** shows the two text field called **`t1`** and **`t2`** (where the user can type text), the combo box **`cb`** (where the user can select one option from a menu) and a button. Use a grid layout manager to position the four components. For example:



The user can type in the first text field the name of a new bank customer and can type in the second text field an amount of money for the new account. The combo box offers only two menu options: **"credit account"** and **"student account"**. For example:



When the user then clicks on the button, the action listener of the button must read the name of the new customer that was typed in the first text field (using the **getText** method of the text field), read the amount of money that was typed in the second text field (using again the **getText** method), and read which menu option was selected in the combo box (using the **getSelectedIndex** method of the combo box, which returns the integer **0** or **1** depending on which menu option the user selected in the combo box), and calls the **create** method of the controller with these two strings and the integer as arguments. The **create** method of the controller then returns a string as result. If the string returned by the **create** method of the controller is different from the empty string **""** then this string must be displayed back to the user using a message dialog (using the **showMessageDialog** method of the **JOptionPane** class). If the string returned by the **create** method of the controller is equal to the empty string **""** then nothing happens in **ViewCreate**.

The **update** method of the **ViewCreate** class does nothing, because the **ViewCreate** class does not graphically display any data from the bank (the model).

Also create a **ControllerCreate** class that extends **Controller** and has the following UML specification:

```
+-----------------------------------------------------------+
|                     ControllerCreate                      |
+-----------------------------------------------------------+
+-----------------------------------------------------------+
| + ControllerCreate(Bank m)                                |
| + create(String name, String amount, int type): String    |
+-----------------------------------------------------------+
```

The **create** method takes as arguments the name of a new bank customer, an amount of money (as a string), and an integer representing the type of bank account to create (where the integer **0** means a credit account and the integer **1** means a student account). The **create** method of the controller then transforms the amount of money from a string to an integer (using the **Integer.parseInt** static method), creates an object from the correct class (based on the type of account specified by the user: credit account or student account) and calls the **addAccount** method of the bank to add the new account object to the bank.

- If no exception occurs then the **create** method of the controller returns the empty string.

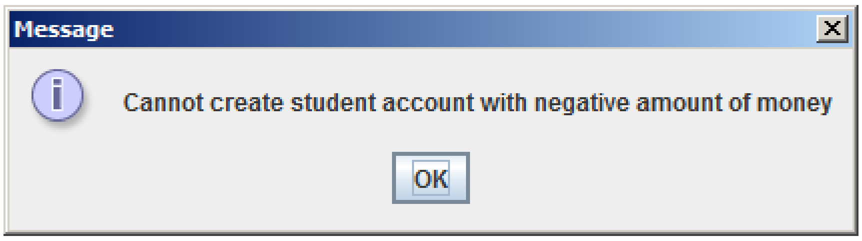
- If the constructor of the **`StudentAccount`** class throws a **`NotEnoughMoneyException`** then the **`create`** method of the controller must catch this exception and return as result the error message from the exception object.
- If the **`parseInt`** method of the **`Integer`** class throws a **`NumberFormatException`** (because the user typed something which is not an integer) then the **`create`** method of the controller must catch this exception and return as result the error message from the exception object.

Modify the **`run`** method of the **`GUI`** class to add a **`ViewCreate`** view that uses a **`ControllerCreate`** controller and the same model as before (not a new model!) Do not delete the previous views.

Note: if at the end of Question 7 you had manually added to your bank (model object) some student accounts and credit accounts for testing, then you must now remove those accounts from the **`run`** method of the anonymous class inside the **`GUI`** class. You do not need these test accounts anymore because you have now a graphical user interface to create new accounts!

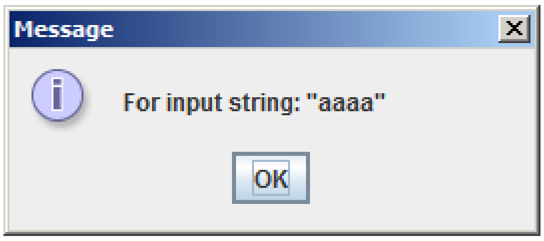
Run your GUI and check that you can correctly use the new view to create different customers for your bank, with different types of accounts.

- Check that, when you create a new account, the simple view is automatically correctly updated to show the new total amount of money in all the bank accounts of the bank.
- Also use the “get money” window of your software to check that the accounts are correctly created with the correct names and correct amounts of money.
- Also check that trying to create a student account with a negative amount of money correctly shows an error message. For example:



Cannot create student account with negative amount of money

- Also check that trying to create an account with an amount of money which is not an integer correctly shows an error message (do not worry about the content of the error message). For example:



For input string: "aaaa"

- After you created a new account, you can also check whether it is a credit account or a student account by using the “withdraw” window to withdraw an amount of money which is bigger than the amount that the account has:
    - if the account you created is a credit account then trying to withdraw from the account an amount of money which is bigger than the amount that the account has will work and the amount in the account will become negative (you can then check that using the “get money” window);
    - if the account you created is a student account then trying to withdraw from the account an amount of money which is bigger than the amount that the account has will fail with an error message and the amount in the account will not change (you can then check that using the “get money” window).

Question 12

We now want to add a new "history" view that allows the user of the software to keep track of how the total amount of money in all the bank accounts of the bank changes over time.

Before we can add such a view to the GUI, first we need to change the model (the **`Bank`** class) to keep track of how the total amount of money in all the bank accounts of the bank changes over time. Therefore, in the **`Bank`** class, add a new private instance variable called **`history`** which is an arraylist of integers. This arraylist must be initialized to contain only one value: zero (meaning that, when the bank is created, it has no money).

We know that the data of the bank can change only in two methods of the **`Bank`** class: in the **`addAccount`** method and in the **`withdraw`** method (this is why these two methods both call **`notifyListeners`**: to tell the views that data has changed and that the views must update themselves). Therefore it is in these two methods **`addAccount`** and **`withdraw`** that we must keep track of how the total amount of money in all the bank accounts of the bank changes over time. Therefore, in these two methods **`addAccount`** and **`withdraw`**, call the **`totalMoney`** method and add the result to the **`history`** arraylist.

Note: in each of the two methods **`addAccount`** and **`withdraw`**, you must call the **`totalMoney`** method and add the result to the **`history`** arraylist before **`notifyListeners`** is called, otherwise the "history" view that you are going to create below will not show the correct results when it is notified by the bank that it must update itself!

Also add to the **`Bank`** class a **`getHistory`** method that returns as result the arraylist of integers which is the bank's history.

Create a **`HistoryPanel`** class that extends **`JPanel`**. The constructor of **`HistoryPanel`** takes as argument a model object of type **`Bank`**, which you need to store in some private instance variable. Add to the **`HistoryPanel`** class two private methods called **`historyMax`** and **`historyMin`** that take an arraylist of integers as argument and return as result the maximum and minimum number in the arraylist, respectively (you can assume that the arraylist contains at least one number). Then add to the **`HistoryPanel`** class a private method called **`historyRange`** that takes an arraylist of integers as argument and returns as result the difference between the max and min of the integers in the arraylist, or returns as result `100` if the difference between the man and min of the integers in the arraylist is strictly less than `100`.

Override the **`protected void paintComponent(Graphics g)`** method inherited from **`JPanel`**, and, inside your new **`paintComponent`** method, draw graphically how the total amount of money in all the bank accounts of the bank changes over time, as follows:

- Compute the following variables (where `history` is the result of calling the **`getHistory`** method of the model):
  ```
  int min = historyMin(history);
  int range = historyRange(history);
  int maxX = getWidth() - 1;
  int maxY = getHeight() - 1;
  int zero = maxY + min * maxY / range;
  ```
- Draw a blue line between the point `(0, zero)` and the point `(maxX, zero)` (this blue line then represents the horizontal "zero" axis).
- For each value `v` at index `i` in the `history` arraylist that you want to draw:
  - Use `x = 10 * i` for the horizontal coordinate;
  - Use `y = zero - v * maxY / range` for the vertical coordinate;
  - Draw red lines between all the points `(x, y)` (if there is only one value in the arraylist then just draw a rectangle of size 1 by 1 at position `(x, y)`).

Create a **ViewHistory** class that extends **View<ControllerHistory>** and has the following UML specification:

```
+-----------------------------------------------+
|                  ViewHistory                  |
+-----------------------------------------------+
+-----------------------------------------------+
| + ViewHistory(Bank m, ControllerHistory c)    |
| + update(): void                              |
+-----------------------------------------------+
```

The **ViewHistory** shows only a **HistoryPanel** object, nothing else. The **update** method of the **ViewHistory** class calls Swing's **repaint** method (this forces Swing to redraw everything every time the model changes, which in turn forces Swing to automatically call the **paintComponent** method of the **HistoryPanel** to redraw the updated version of the bank's history).

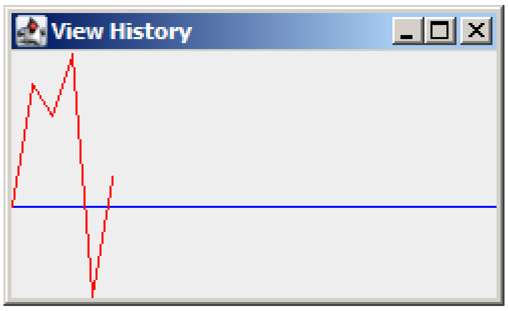
Also create a **ControllerHistory** class that extends **Controller** and has the following UML specification:

```
+-------------------------------------------------------------+
|                      ControllerHistory                      |
+-------------------------------------------------------------+
+-------------------------------------------------------------+
| + ControllerHistory(Bank m)                                 |
+-------------------------------------------------------------+
```

Since the **ViewHistory** does not receive any input from the user, the **ControllerHistory** does nothing. (Note: since **ControllerHistory** does nothing anyway, we could just remove it and replace it with **Controller** in the definition of **ViewHistory** and in the **run** method of the **GUI** class, but here we keep **ControllerHistory** just to make the Model-View-Controller design pattern very clear.)

Modify the **run** method of the **GUI** class to add a **ViewHistory** view that uses a **ControllerHistory** controller and the same model as before (not a new model!) Do not delete the previous views.

Run your GUI and check that adding new bank customers and withdrawing money from customers correctly updates the graphical history of the bank's total amount of money. For example, if the user of the software creates a student account with 2000 yuan, then withdraws 500 yuan from the student account, then creates a credit account with 1000 yuan, then withdraws 4000 yuan from the credit account, then creates a student account with 2000 yuan, then the graphical history of the bank's total amount of money must look like this:



Check that all the other features of your software still work correctly. Also run your tests and the CLI to make sure everything still works correctly.

Question 13

We now want to store the bank's information (accounts and history, but not the bank's name and not the bank's listeners) into a binary file using object serialization.

Add to the **Bank** class a new private **file** instance variable. In the constructor, initialize the **file** instance variable to be a **File** object for a binary file named **"XXX.bin"** (where **XXX** is replaced with the name of the bank). Add to the **Bank** class a new public method called **saveData** that takes no argument, returns nothing, and uses object serialization to save into the binary file the **accounts** and **history** arraylists of the bank. Modify the constructor of the **Bank** class to read all the information from this binary file, if it exists, and put it into the corresponding arraylists (if the binary file does not exist then the arraylists must be initialized as before).

Make other classes implement Java's **Serializable** interface as appropriate.

Add to the **Controller** superclass a protected **shutdown** method that:

- calls the **saveData** method of the model;
- manually terminates the program using **System.exit(0)**.

Then modify the **View** superclass to:

- hide the frame when the user clicks on the "close" button;
- add a "window closing" event handler (use an anonymous window adapter) that calls the controller's **shutdown** method.

Run your GUI, add some accounts, make some withdrawals, exit the software, run the GUI again, and check that all the account and history information is still there and is correct. Also check that the binary file is correctly created inside the folder for your Eclipse project (it is a binary file though, so you cannot read its content).

Note that the command line interface that you created in Question 6 reads all the information saved into the binary file when you start the command line interface (because the constructor of the **Bank** class will automatically read the content of the binary file if it exists and has the correct name) but it does not write the information back into the binary file when you quit the command line. This is easy to fix: in the **main** method of the **CLI** class, just before printing the **"Goodbye!"** message, call the **saveData** method of the **Bank** object that you created at the start of the **main** method.

Run the command line interface and check that any change you make to the bank's information using the command line interface is correctly saved into the binary file when you quit the command line interface. You can run the GUI to check this (but do not run the GUI and the CLI at the same time!)

Use the **Test** class to run all the tests for the software and check that all the tests still work (delete the binary file before you do this, otherwise the **Bank** constructor that you use in your tests will read the account data from the binary file, which will change the results of some of your tests).

The end. That was fun!