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 a book lending system for a Library. The system has different roles for registered users. There are two types of user roles: borrower and lender.

Write an **`IUser`** interface for library users, with the following UML specification:

```
+-----------------------------------+
|          <<interface>>            |
|              IUser                |
+-----------------------------------+
| + getName(): String               |
| + getBook(): int                  |
| + moreBook(int number): void      |
+-----------------------------------+
```

and a **`User`** class that implements **`IUser`** and has the following UML specification:

```
+-------------------------------------+
|                User                 |
+-------------------------------------+
| - name: String                      |
| - book: int                         |
+-------------------------------------+
| + User(String name, int book)       |
| + getName(): String                 |
| + getBook(): int                    |
| # setBook(int book): void           |
| + moreBook(int number): void        |
| + testUser(): void                  |
+-------------------------------------+
```

The **`name`** instance variable indicates the user name. The **`book`** instance variable indicates the number of books *borrowed* by the user.

The **setBook** method changes the number of books borrowed by the user. The **setBook** method is **protected**, not **public**. This means that only subclasses of the **User** class can use the **setBook** method. All the other classes in the system cannot use the **setBook** method, so they cannot change the number of books borrowed by a user.

The purpose of the **moreBook** method is to increase the number of books *borrowed* or *lent* by the user (depending on what kind of user it is) by the number given as argument to the method. The **moreBook** method of the **User** class is **abstract**, since we do not know what kind of role the user is (a borrower borrows books from other users and a lender lend books to other users).

Also add to your program a **Test** class to test your **User** class.

```java
public class Test {

	public static void main(String[] args) {
		User.testUser();
	}
}
```

Question 2

Add a class **Lender** that extends **User**. The constructor of the **Lender** class takes as arguments a name and the number of books *lent* by the user. The **Lender** class does not have any instance variable.

Warning: the constructor of the **Lender** class takes as argument the number of books *lent* by the lender, but the **book** instance variable of the **User** class indicates how many books the user *borrowed*! Lending books is the same as borrowing a *negative* number of books.

The **moreBook** method of the **Lender** class increases the number of books *lent* by the lender user by the number given as argument to the method (so the books *borrowed* by the lender becomes more negative!)

Make sure you test all the methods of your new **Lender** class using **testLender** method.

Here are some tests for your new **Lender** class:

```java
public static void testLender()
	{
		Lender l = new Lender("Anna",5);
		System.out.println(l.getName() == "Anna");
		System.out.println(l.getBook() == -5);
		l.setBook(-6);
		System.out.println(l.getBook() == -6);
		l.moreBook(2);
		System.out.println(l.getBook() == -8);
		l.moreBook(-9);
		System.out.println(l.getBook() == 1);
	}
```

Also, do not forget to modify your **Test** class to test the **Lender** class.

Question 3

Add a class **Borrower** that extends **User**. The constructor of the **Borrower** class takes a name and a number of books *borrowed* by the borrower. If the number of books given as argument is strictly less than zero, then the constructor must throw a **NotALenderException** with the message **"A new borrower cannot lend books."**. The **borrower** class does not have any instance variable.

The **moreBook** method of the **Borrower** class increases the number of books *borrowed* by the borrower by the number of books given as argument to the method (so the books *borrowed* by the borrower becomes more positive!)

For example, if a borrower currently borrows 10 books and **`moreBook(2)`** is called then the borrower borrows 12 books. It is fine for the **`moreBook`** method to be given a negative value as argument, which means the borrower then just returned some books. For example, if a borrower currently borrows 10 books and **`moreBook(-2)`** is called then the borrower borrows 8 books.

However, a borrower cannot lend books, so the number of books borrowed by the borrower must always be positive or zero, never negative. If the argument given to the **`moreBook`** method is too negative and would change the book variable into a negative value, then the number of books borrowed by the borrower must not change and the **`moreBook`** method must throw a **`NotALenderException`** with the message "**`A borrower cannot lend XXX book(s).`**", where **`XXX`** is replaced with the result of **`-(book + number)`**.

For example, if a borrower currently borrows 10 books and **`moreBook(-12)`** is called then the borrower still borrows 10 books and the method throws a **`NotALenderException`** with the message "**`A borrower cannot lend 2 book(s).`**".

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

Change other classes and interfaces as necessary.

Make sure you test your new **`Borrower`** class, including inherited methods. Here are some tests for your new **`Borrower`** class:

```
public static void testBorrower()
	{
		try
		{
			Borrower b = new Borrower("Bob",-1);
		} catch(NotALenderException e)
		{
		System.out.println(e.getMessage().equals("A new borrower cannot lend
books."));
		}
		try
		{
			Borrower b = new Borrower("Bob",10);
			System.out.println(b.getName()=="Bob");
			System.out.println(b.getBook()==10);
			b.setBook(5);
			System.out.println(b.getBook()==5);
			b.moreBook(2);
			System.out.println(b.getBook()==7);
			b.moreBook(-2);
			System.out.println(b.getBook()==5);
			b.moreBook(-5);
			System.out.println(b.getBook()==0);
			b.moreBook(-1);
		} catch(NotALenderException e)
		{
		System.out.println(e.getMessage().equals("A borrower cannot lend 1
book(s)."));
		}

	}
```

Also, do not forget to modify your **`Test`** class to test the **`Borrower`** class.

Question 4

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

```
+----------------------------------------+
|                 Library                |
+----------------------------------------+
| - name: String                         |
| - users: ArrayList<IUser>              |
+----------------------------------------+
| + Library(String name)                 |
| + addUser(IUser user): void            |
| + totalBorrowedBooks(): int            |
| + getBook(String name): int            |
| + moreBook(String name, int number): void |
| + testLibrary(): void                  |
+----------------------------------------+
```

When a library is created, it has an arraylist of users (`IUser`) but the arraylist is empty (the arraylist does not contain any user).

The **`addUser`** method takes a user (`IUser`) as argument and adds the user to the arraylist of users for the library.

The **`totalBorrowedBooks`** method returns as result the total number of books borrowed by all users of the library (the result can be either positive or negative).

The **`getBook`** method takes as argument the name of a user and returns as result the number of books currently borrowed by the user. If the library does not have a user with the given name, then the **`getBook`** method must throw an **`UnknownUserException`** with the message **`"User XXX unknown."`**, where **`XXX`** is replaced with the name of the user. Do not worry about multiple users having the same name. You can assume all user names are unique in the arraylist.

The **`moreBook`** method takes as argument the name of a user and a number of books and changes the number of books currently borrowed by that user. If the library does not have a user with the given name, then the **`moreBook`** method must throw an **`UnknownUserException`** with the message **`"User XXX unknown."`**, where **`XXX`** is replaced with the name of the user. Do not worry about multiple users having the same name.

Note: the **`moreBook`** 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 **`Library`** class in **`testLibrary`** method. Here are some tests for your new **`Library`** class:

```java
public static void testLibrary() {
		Library li = new Library("UIC Library");

		System.out.println(li.totalBorrowedBooks() == 0);
		li.addUser(new Lender("L1", 10));
		try {
			System.out.println(li.getBook("L1") == -10);
			System.out.println(li.totalBorrowedBooks() == -10);
			li.addUser(new Borrower("B1", 20));
			System.out.println(li.getBook("L1") == -10);
			System.out.println(li.getBook("B1") == 20);
			System.out.println(li.totalBorrowedBooks() == 10);
			li.getBook("B2");
		} catch(UnknownUserException  ex) {
			System.out.println(ex.getMessage().equals("User B2 unknown."));
		} catch(NotALenderException ex) {
			// This should never happen!
			System.out.println(false);
		}
```

```
            //More test cases are needed…
             ……
              ……
              ……
    }
```

Also, do not forget to modify your **Test** class to test the **Library** class.

Question 5

In this question and the next one we want to create a command line interface (CLI) for our Library book lending system.

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 system 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 **Library** object with the name **"UIC Library"**. The **main** method of the **CLI** class must then print a menu that allows the user of your system to do six different actions that involve the library 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 get:3 more:4 less: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 get:3 more:4 less: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 library 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 get:3 more:4 less:5 quit:6): 7
Unknown action!
Type an action (total:1 add:2 get:3 more:4 less: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 get:3 more:4 less:5 quit:6): aaaa
You must type an integer!
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): bbbb
You must type an integer!
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6):
```

Here are the detailed explanations for each action.

Action 1: printing the total number of books borrowed by all users.

When the user of the system specifies action 1, your program must simply print on the screen the total number of books currently borrowed by all users of the library. 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 get:3 more:4 less:5 quit:6): 1
Total number of borrowed books: 50
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6):
```

Action 2: adding a new user to the library.

When the user of the software specifies action 2, your program must add a new user to the library. To add a new user, your program needs to ask the user three things: the role of user (an integer read using **`readPosInt`**: the integer **1** represents *lender*, the integer **2** represents *borrower*, any other integer must result in an error message **`"Unknown`**

user role!" being printed and the software going immediately back to the main menu), the name of the user (a string read using **readLine**), and the initial number of books that the user lends (for a lender) or borrows (for a borrower). You program must then create the correct user, add it to the library, and print an information message. The program then goes back to the menu.

For example (where **2**, **3**, **2**, **1**, **Anna**, **5**, **2**, **2**, **Bob**, and **10** are inputs from the user):

```
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 2
Type the user role (lender:1 borrower:2): 3
Unknown user role!
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 2
Type the user role (lender:1 borrower:2): 1
Enter the name of the user: Anna
Enter the initial number of borrowed books: 5
Lender "Anna" lending 5 book(s) has been added.
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 2
Type the user role (lender:1 borrower:2): 2
Enter the name of the user: Bob
Enter the initial number of borrowed books: 10
Borrower "Bob" borrowing 10 book(s) has been added.
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6):
```

Note that the **readPosInt** method prevents the initial number of books from being negative, so the constructor for the **Borrower** class will never throw a **NotALenderException** when you create a borrower 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: get the number of books borrowed by a given user.

When the user of the system specifies action 3, your program must ask the user to type the name of a user, and the program then prints the number of books which is currently borrowed by this user.

For example (where 3, Anna, 3, and Bob are inputs from the user):

```
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 3
Enter the name of the user: Anna
Anna borrows -5 book(s).
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 3
Enter the name of the user: Bob
Bob borrows 10 book(s).
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6):
```

If the name of the user is wrong, then an **UnknownUserException** exception will be thrown by the **Library** 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 get:3 more:4 less:5 quit:6): 3
Enter the name of the user: aaaa
User aaaa unknown.
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6):
```

Action 4: increasing the number of books of a given user.

When the user of the software specifies action 4, your program must ask the user to type the name of a user, and a number of books, and the program then uses that number to increase the number of books lent or borrowed by the user. Then the program goes back to the main menu.

For example:

```
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 3
Enter the name of the user: Anna
Anna borrows -5 book(s).
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 4
Enter the name of the user: Anna
Enter the number of books: 2
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 3
Enter the name of the user: Anna
Anna borrows -7 book(s).
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 3
Enter the name of the user: Bob
Bob borrows 10 book(s).
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 4
Enter the name of the user: Bob
Enter the number of books: 2
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 3
Enter the name of the user: Bob
Bob borrows 12 book(s).
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6):
```

If the name of the user is wrong, then an **UnknownUserException** exception will be thrown by the **Library** 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 2 are inputs from the user):

```
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 4
Enter the name of the user: aaaa
Enter the number of books: 2
User aaaa unknown.
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6):
```

Note that, even if a consumer is a borrower, the **readPosInt** method prevents the typed number of books from being negative. This means a borrower will never throw a **NotALenderException**. 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)**.

For example (where 3, Bob, 4, Bob, and -15 are inputs from the user):

```
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 3
Enter the name of the user: Bob
Bob borrows 12 book(s).
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 4
Enter the name of the user: Bob
Enter the number of books: -15
Positive integers only!
Enter the number of books:
```

Action 5: decreasing the number of books of a given user.

When the user of the software specifies action 5, your program must ask the user to type the name of a user, and a number of books, and the program then uses that number to decrease the number of books *lent* or *borrowed* by the user. Then the program goes back to the main menu.

Note: the library object that you are using does not have a method to decrease books. So, in the code of the **main** method of the **CLI** class, simulate decreasing books by simply increasing books by a negative number! For example, decreasing the number of books of a user by 5 books is the same as increasing the number of books by -5 books.

For example:

```
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 3
```

```
Enter the name of the user: Anna
Anna borrows -7 book(s).
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 5
Enter the name of the user: Anna
Enter the number of books: 6
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 3
Enter the name of the user: Anna
Anna borrows -1 book(s).
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 3
Enter the name of the user: Bob
Bob borrows 12 book(s).
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 5
Enter the name of the user: Bob
Enter the number of books: 5
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 3
Enter the name of the user: Bob
Bob borrows 7 book(s).
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6):
```

If the name of the user is wrong, then an **`UnknownUserException`** exception will be thrown by the **`Library`** 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 2 are inputs from the user):

```
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 5
Enter the name of the user: aaaa
Enter the number of books: 2
User aaaa unknown.
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6):
```

If a consumer is a borrower and the number of books typed by the user is too big, then a **`NotALenderException`** exception will be thrown by the **`Borrower`** 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, `Bob`, 5, `Bob`, `10`are inputs from the user):

```
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 3
Enter the name of the user: Bob
Bob borrows 7 book(s).
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 5
Enter the name of the user: Bob
Enter the number of books: 10
A borrower cannot lend 3 book(s).
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 3
Enter the name of the user: Bob
Bob borrows 7 book(s).
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6):
```

Action 6: quitting the program.

When the user of the system 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 get:3 more:4 less:5 quit:6): 6
Goodbye!
```

Here is a more complete example of running the system:

```
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): aaaa
You must type an integer!
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): -100
Positive integers only!
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 7
Unknown action!
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 1
Total number of borrowed books: 0
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 2
Type the user role (lender:1 borrower:2): 0
Unknown user role!
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 2
Type the user role (lender:1 borrower:2): 3
Unknown user role!
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 2
Type the user role (lender:1 borrower:2): 1
Enter the name of the user: Anna
Enter the initial number of borrowed books: 5
Lender "Anna" lending 5 book(s) has been added.
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 2
Type the user role (lender:1 borrower:2): 2
Enter the name of the user: Bob
Enter the initial number of borrowed books: 10
Borrower "Bob" borrowing 10 book(s) has been added.
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 1
Total number of borrowed books: 5
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 3
Enter the name of the user: Anna
Anna borrows -5 book(s).
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 3
Enter the name of the user: Bob
Bob borrows 10 book(s).
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 3
Enter the name of the user: aaaa
User aaaa unknown.
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 4
Enter the name of the user: Anna
Enter the number of books: 2
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 3
Enter the name of the user: Anna
Anna borrows -7 book(s).
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 3
Enter the name of the user: Bob
Bob borrows 10 book(s).
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 4
Enter the name of the user: Bob
Enter the number of books: 2
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 3
Enter the name of the user: Bob
Bob borrows 12 book(s).
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 3
Enter the name of the user: Anna
Anna borrows -7 book(s).
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 5
Enter the name of the user: Anna
Enter the number of books: 6
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 3
Enter the name of the user: Anna
Anna borrows -1 book(s).
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 3
Enter the name of the user: Bob
Bob borrows 12 book(s).
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 5
Enter the name of the user: Bob
Enter the number of books: 5
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 3
Enter the name of the user: Bob
```

```
Bob borrows 7 book(s).
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 5
Enter the name of the user: Bob
Enter the number of books: 10
A borrower cannot lend 3 book(s).
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 3
Enter the name of the user: Bob
Bob borrows 7 book(s).
Type an action (total:1 add:2 get:3 more:4 less:5 quit:6): 6
Goodbye!
```

Question 7

We now want to create a graphical user interface (GUI) for our library book lending system. Since we want the system 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 **Library** class is the class that contains all the data for the library. Therefore the **Library** class plays the role of the model. Therefore the **Library** class needs to keep an arraylist of model listeners that need to be notified every time the library (the model) changes.

1. Add to the **Library** class an private instance variable, an arraylist of **ModelListener**. When a library is created, it has an empty arraylist of listeners.
2. Also add to the **Library** class an **addListener** method that takes a **ModelListener** as argument and adds it to the arraylist of listeners.
3. Also add to the **Library** class a private **notifyListeners** method that takes nothing as argument and calls the **update** method of all the listeners of the library.
4. Then change the **addUser** and **moreBook** methods so that they call the **notifyListeners** every time a change is made to the library's data (only the **addUser** and **moreBook** methods change the library's data, so only these two methods need to call the **notifyListeners** method; the **totalBorrowedBooks** and **getBook** methods do not change the library'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: Library                                  |
| - c: ControllerSimple                         |
| - label: JLabel                               |
+-------------------------------------------+
| + ViewSimple(Library m, ControllerSimple c) |
| + update(): void                          |
+-------------------------------------------+
```

The constructor of the **ViewSimple** class registers the view with the model (the library) 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 number of books borrowed by all users of the library, 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 number of books borrowed by all users of the library.

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

```
+-----------------------------------------+
|            ControllerSimple             |
+-----------------------------------------+
| - m: Library                            |
+-----------------------------------------+
| + ControllerSimple(Library 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.

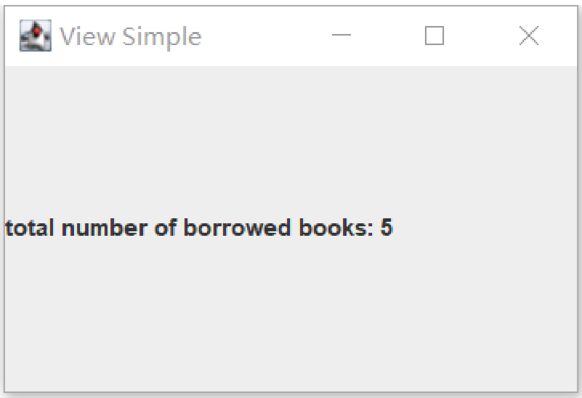
Sixth, we need to connect the model, the view, and the controller to each other. So in the **run** method of the anonymous class:

- create a **Library** object (the model object) with the name **"UIC Library"**;
- 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 number of books borrowed by all users of the library. This total number must be zero, since the library (model object) you just created above does not contain any user!

As a test, in the **run** method of the anonymous class, you can try to manually add to your library (model object) some lenders and borrowers to check that whether your GUI displays the correct the total number of books borrowed by all users in the library.

For example, if we add a lender who has borrowed -5 books and a borrower who has borrowed 10 books, the GUI class running result is shown in the following figure:



total number of borrowed books: 5

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: Library                              |
| # c: T                                    |
+-------------------------------------------+
| + View(Library 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<ControllerSimple>** 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: Library                              |
+-------------------------------------------+
| + Controller(Library 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.

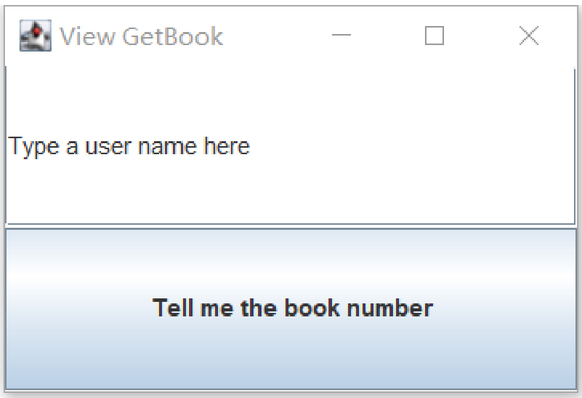
Question 9

We now want to add a new "get book" view that allows the user of the system to check how many books a specific user is borrowing.

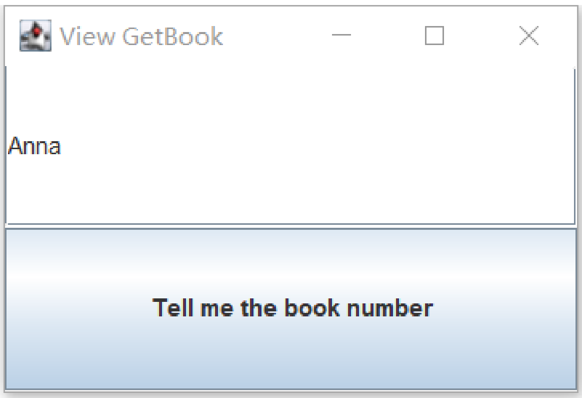
Create a **`ViewGetBook`** class that extends **`View<ControllerGetBook>`** and has the following UML specification:

```
+--------------------------------------------------+
|                   ViewGetBook                     |
+--------------------------------------------------+
| - t: JTextField                                   |
+--------------------------------------------------+
| + ViewGetBook(Library m, ControllerGetBook c)     |
| + update(): void                                  |
+--------------------------------------------------+
```

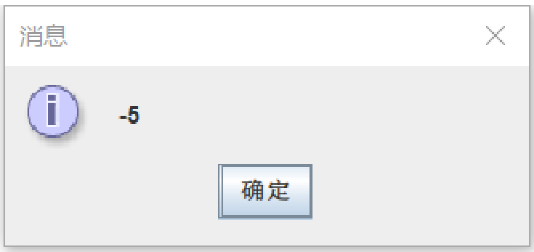
The **`ViewGetBook`** shows the text field called **`t`** (where the user can type text) and a button. Use a grid layout manager to position the two components. For example:



The user can type in the text field **`t`** the name of a library user. For example:



When the user then clicks on the button, the action listener of the button must read the name of the library user that was typed in the text field (using the **`getText`** method of the text field) and must call the **`getBook`** method of the controller with that user name as argument. The **`getBook`** 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:


消息

-5

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

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

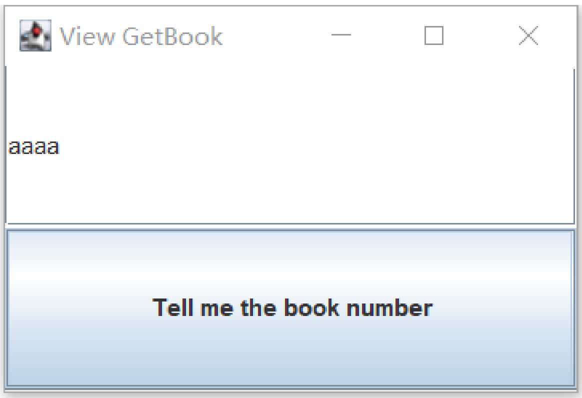
```
+-----------------------------------------+
|            ControllerGetBook            |
+-----------------------------------------+
+-----------------------------------------+
| + ControllerGetBook (Library m)          |
| + getBook(String name): String          |
+-----------------------------------------+
```

The **getBook** method takes the name of a user as argument. The **getBook** method of the controller then calls the **getBook** method of the library to get the number of books currently borrowed by that user. The **getBook** method of the controller then transforms the integer result of the **getBook** method of the library into a string and returns that string as result (to the view). If the **getBook** method of the library throws an **UnknownUserException** then the **getBook** method of the controller must catch this exception and return as result the error message from the exception object.

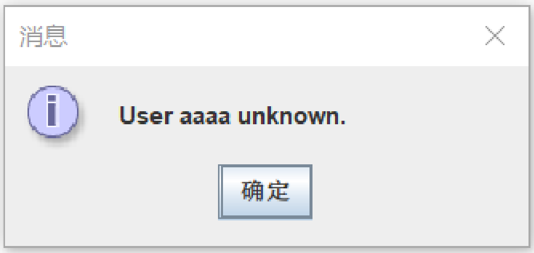
Modify the **run** method of the **GUI** class to add a **ViewGetBook** view that uses a **ControllerGetBook** controller and the same model as before (not a new model!) Do not delete the previous view.

Run your GUI and check that you can correctly use the new view to query the number of books borrowed by different users of your library (obviously your library must have some users in it to test this: see the last paragraph of Question 7).

Also check that querying the number of books of an unknown user correctly shows an error message. For example:


View GetBook

aaaa


消息

User aaaa unknown.

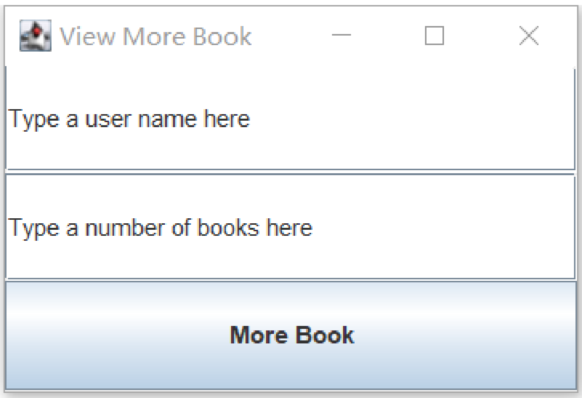
Question 10

We now want to add a new “more book” view that allows the user of the system to increase the number of book *borrowed* or *lent* by the user (depending on what kind of user it is) of a specific user.

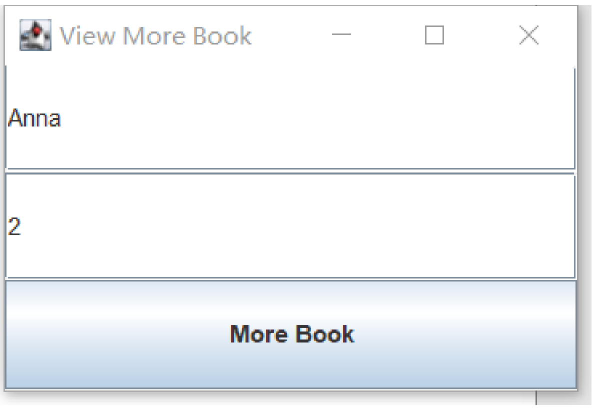
Create a **ViewMoreBook** class that extends **View<ControllerMoreBook>** and has the following UML specification:

```
+--------------------------------------------------+
|                   ViewMoreBook                   |
+--------------------------------------------------+
| - t1: JTextField                                 |
| - t2: JTextField                                 |
+--------------------------------------------------+
| + ViewMoreBook(Library m, ControllerMoreBook c)  |
| + update(): void                                 |
+--------------------------------------------------+
```

The **ViewMoreBook** 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:



The user can type in the first text field the name of a library user and can type in the second text field a number of books. For example:



When the user then clicks on the button, the action listener of the button must read the name of the library user that was typed in the first text field (using the **getText** method of the text field) and the number of books that was typed in the second text field (using again the **getText** method) and must call the **moreBook** method of the controller with these two strings as arguments. The **moreBook** method of the controller then returns a string as result. If the string returned by the **moreBook** 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 **moreBook** method of the controller is equal to the empty string **""** then nothing happens in **ViewMoreBook**.

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

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

```
+-------------------------------------------------+
|                ControllerMoreBook               |
+-------------------------------------------------+
+-------------------------------------------------+
| + ControllerMoreBook(Library m)                 |
| + moreBook(String name, String number): String |
+-------------------------------------------------+
```

The **`moreBook`** method takes the name of a user and a number of books (as a string) as arguments. The **`moreBook`** method of the controller then transforms the number of books from a string to an integer (using the **`Integer.parseInt`** static method) and calls the **`moreBook`** method of the library to increase the number of books *borrowed* or *lent* by the user (depending on what kind of user it is) of a specific user, by the given argument.

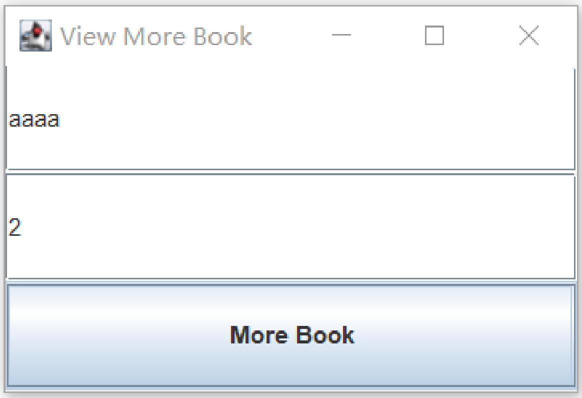
- If no exception occurs then the **`moreBook`** method of the controller returns the empty string.
- If the **`moreBook`** method of the library throws an **`UnknownUserException`** then the **`moreBook`** method of the controller must catch this exception and return as result the error message from the exception object.
- If the **`moreBook`** method of the library throws a **`NotALenderException`** then the **`moreBook`** 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 **`moreBook`** 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 system to increase the number of books of a user by a negative number, so there is no need to check for that.

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

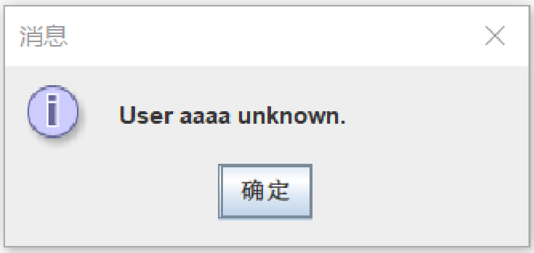
Run your GUI and check that you can correctly use the new view to increase the number of books for different users of your library (obviously your library must have some users in it to test this: see the last paragraph of Question 7).

- Check that, when you increase a user's book, the **simple view** is **automatically** correctly updated to show the new total number of borrowed books for all users of the library.
- Also use the "get book" view to check that the user's book value correctly changed.
- Also check that increasing the book number of an unknown user correctly shows an error message. For example:


View More Book

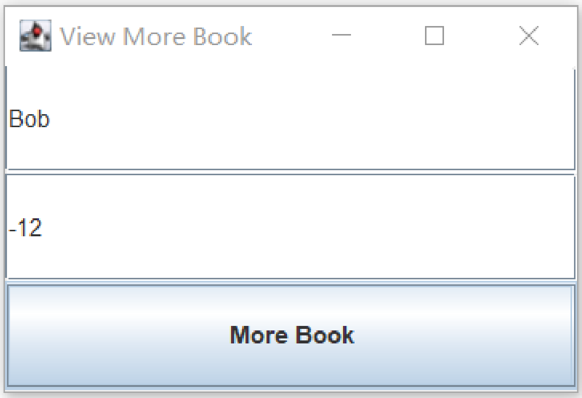
aaaa

2


消息

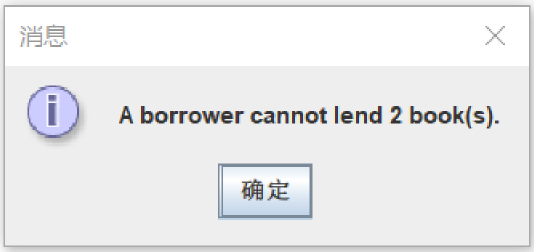
User aaaa unknown.

- Also check that increasing the book of a user by a large negative number correctly shows an error message. For example:


View More Book

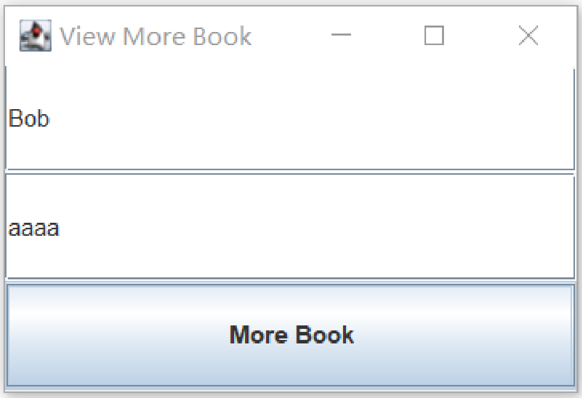
Bob

-12


消息

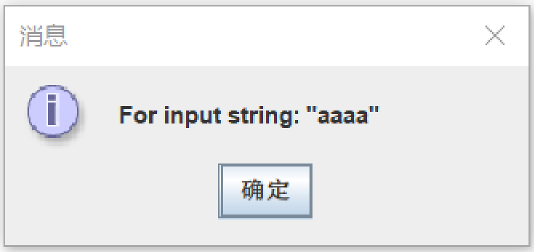
A borrower cannot lend 2 book(s).

- Also check that trying to increase the book of a user by a number which is not an integer correctly shows an error message (do not worry about the content of the error message). For example:


View More Book

Bob

aaaa


消息

For input string: "aaaa"

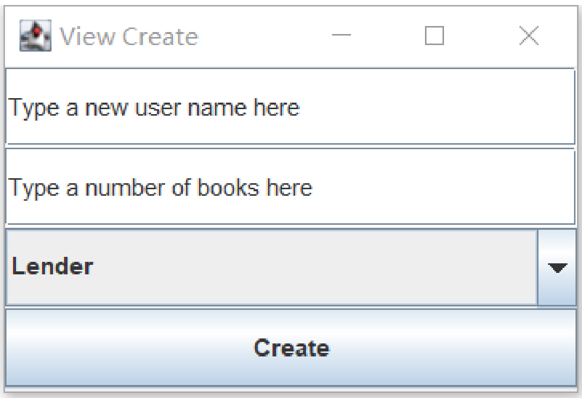
Question 11

We now want to add a new “create” view that allows the user of the system to create a user for the library system.

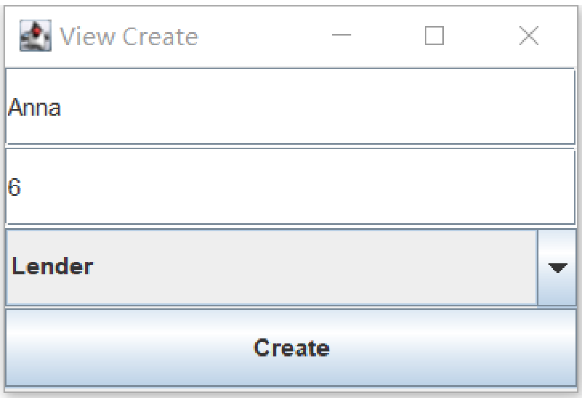
Create a **ViewCreate** class that extends **View<ControllerCreate>** and has the following UML specification:

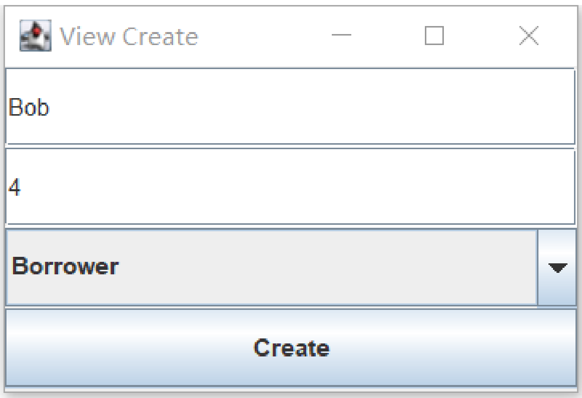
```
+-----------------------------------------------+
|                  ViewCreate                   |
+-----------------------------------------------+
| - t1: JTextField                              |
| - t2: JTextField                              |
| - cb: JComboBox<String>                       |
+-----------------------------------------------+
| + ViewCreate(Library 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 library user and can type in the second text field a number of books for the new library user. The combo box offers only two menu options: **"Lender"** and **"Borrower"**. For example:





When the user then clicks on the button, the action listener of the button must read the name of the new library user that was typed in the first text field (using the **getText** method of the text field), read the number of books 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 library (the model).

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

```
+-------------------------------------------------------------+
|                      ControllerCreate                       |
+-------------------------------------------------------------+
+-------------------------------------------------------------+
| + ControllerCreate(Library m)                               |
| + create(String name, String number, int type): String      |
+-------------------------------------------------------------+
```

The **create** method takes as arguments the name of a new library user, a number of books (as a string), and an integer representing the role of user to create (where the integer **0** means a lender and the integer **1** means a borrower). The **create** method of the controller then transforms the book number from a string to an integer (using the **`Integer.parseInt`** static method), creates an object from the correct class (based on the role specified by the user input: lender or borrower) and calls the **`addUser`** method of the library to add the new user object to the library.

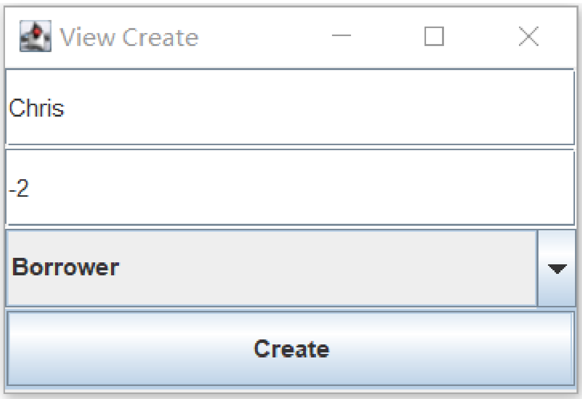
- If no exception occurs then the **`create`** method of the controller returns the empty string.
- If the constructor of the **`Borrower`** class throws a **`NotALenderException`** then the **`create`** method of the controller must catch this exception and return as result the error message from the 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 library (model object) some users for testing, then you must now remove those users from the **`run`** method of the anonymous class inside the **`GUI`** class. You do not need these test users anymore because you have now a graphical user interface to create new users!

Run your GUI and check that you can correctly use the new view to create different users for your library, with different types of roles.

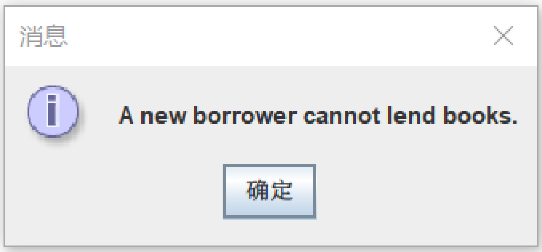
- Check that, when you create a new user, the simple view is automatically correctly updated to show the new total number of books borrowed by all users.
- Also use the "get book" view to check that the users are correctly created with the correct names and correct number of books.
- Also check that trying to create a borrower with a negative number of books correctly shows an error message. For example:



Chris

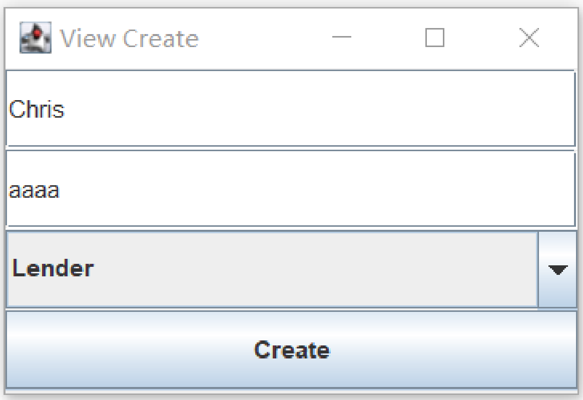
-2

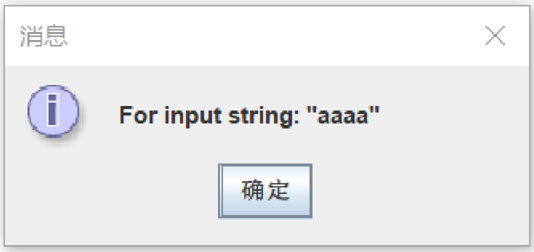
Borrower



A new borrower cannot lend books.

- Also check that trying to create a user with a number of books 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 user, you can also check whether it is a lender or a borrower using the “more book” view to increase the number of books of the user by a big negative number:
  - if the new user you created is a lender, then increasing the number of books by a big negative value will work and the number of books borrowed by the user will just become a larger value (you can then check that using the “get book” view);
  - if the new user you created is a lender, then increasing the number of books by a big negative value will fail with an error message and the number of books borrowed by the user will not change (you can then check that using the “get book” view).

Question 12

We now want to add a new “history” view that allows the user of the system to keep track of how the total number of books borrowed by all users of the library changes over time.

Before we can add such a view to the GUI, first we need to change the model (the **`Library`** class) to keep track of how the total number of books borrowed by all users of the library changes over time. Therefore, in the **`Library`** 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 library is created, it has no books being lent or borrowed).

We know that the data of the library can change only in two methods of the **`Library`** class: in the **`addUser`** method and in the **`moreBook`** 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 that we must keep track of how the total number of books borrowed by all users of the library changes over time. Therefore, in these two methods, call the **`totalBorrowedBooks`** method and add the result to the **`history`** arraylist.

Note: in each of the two methods **`addUser`** and **`moreBook`**, you must call the **`totalBorrowedBooks`** method and add the result to the **`history`** arraylist <u>before</u> **`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 library that it must update itself!

Also add to the **`Library`** class a **`getHistory`** method that returns as result the arraylist of integers which is the library’s history.

Create a **`HistoryPanel`** class that extends **`JPanel`**. The constructor of **`HistoryPanel`** takes as argument a model object of type **`Library`**, 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 **10** if the difference between the man and min of the integers in the arraylist is strictly less than **10**.

Override the **protected void paintComponent(Graphics g)** method inherited from **JPanel**, and, inside your new **paintComponent** method, draw graphically how the total number of books borrowed by all users of the library 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(Library 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 history).

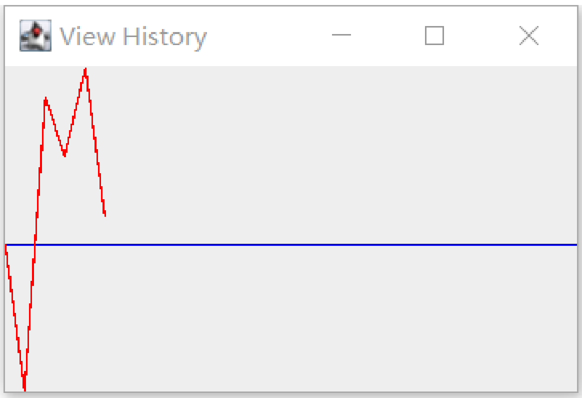
Also create a **ControllerHistory** class that extends **Controller** and has the following UML specification:

```
+------------------------------------------------------------+
|                    ControllerHistory                       |
+------------------------------------------------------------+
+------------------------------------------------------------+
| + ControllerHistory(Library 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 users and changing the books borrowed by users correctly updates the graphical history of the library's total number of borrowed books. For example, if the user of the system creates a lender with 5 lent books, then create a borrower with 10 borrowed books, then increase the books of the lender by 2 books, increase the books of the borrower by 3 books and increase the books of the borrower by -5 books, the plotted history is shown as follows:



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