Lab 9

Demo Points

1. Story 32 (5 Marks)
2. Story 33 (5 Marks)
3. Story 34 (5 Marks)
4. Story 35 (5 Marks)
5. Story 36 (5 Marks)
6. Story 37 (5 Marks)

Next Week

Next week, you will be working on the Library program again. We will be adding the ability to enter new friends, items, and loans from the keyboard.

To do this, we will create a new class that has a 1-1 relationship with MyLibrary. We do this to increase the cohesion of the MyLibrary class. Classes should do one thing so they are easy to understand. MyLibrary manages the program. It is the main entry point, so we want to keep it focused on that task. Input and Output are complicated activities, so we co not want to clutter our MyLibrary class with those responsibilities. Figure 1 show the new design with the added class LibraryIO.

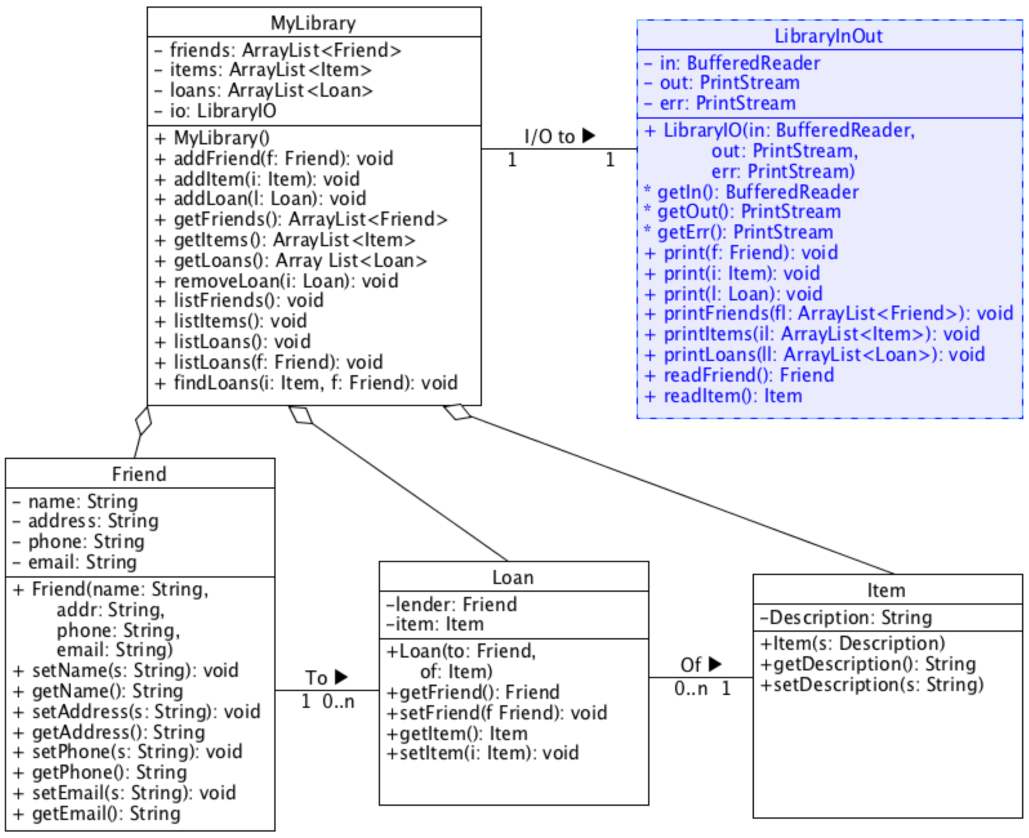

MyLibrary
- friends: ArrayList<Friend>
- items: ArrayList<Item>
- loans: ArrayList<Loan>
- io: LibraryIO
+ MyLibrary()
+ addFriend(f: Friend): void
+ addItem(i: Item): void
+ addLoan(l: Loan): void
+ getFriends(): ArrayList<Friend>
+ getItems(): ArrayList<Item>
+ getLoans(): Array List<Loan>
+ removeLoan(i: Loan): void
+ listFriends(): void
+ listItems(): void
+ listLoans(): void
+ listLoans(f: Friend): void
+ findLoans(i: Item, f: Friend): void
I/O to ▶
1
1
LibraryInOut
- in: BufferedReader
- out: PrintStream
- err: PrintStream
+ LibraryIO(in: BufferedReader, out: PrintStream, err: PrintStream)
* getIn(): BufferedReader
* getOut(): PrintStream
* getErr(): PrintStream
+ print(f: Friend): void
+ print(i: Item): void
+ print(l: Loan): void
+ printFriends(fl: ArrayList<Friend>): void
+ printItems(il: ArrayList<Item>): void
+ printLoans(ll: ArrayList<Loan>): void
+ readFriend(): Friend
+ readItem(): Item
Friend
- name: String
- address: String
- phone: String
- email: String
+ Friend(name: String, addr: String, phone: String, email: String)
+ setName(s: String): void
+ getName(): String
+ setAddress(s: String): void
+ getAddress(): String
+ setPhone(s: String): void
+ getPhone(): String
+ setEmail(s: String): void
+ getEmail(): String
To ▶
1 0..n
Loan
-lender: Friend
-item: Item
+Loan(to: Friend, of: Item)
+getFriend(): Friend
+setFriend(f Friend): void
+getItem(): Item
+setItem(i: Item): void
Of ▶
0..n 1
Item
-Description: String
+Item(s: Description)
+getDescription(): String
+setDescription(s: String)

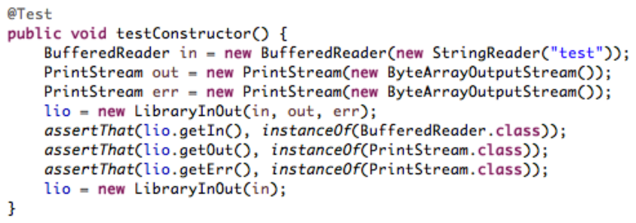
Figure 1: Design altered to include new LibraryInOut class (in blue)

This class will hold the standard input, standard output, and standard error for our program. We include them in the class instead of relying on System.in, System.out, and System.err, so we can easily change them. For example, for testing, we can put in fake input and output. As another example, we we start working on the Graphical User Interface, we can use the same input and output streams to print to windows in our GUI using the same methods.

In the new class the '*' character indicates that the method is protected. That is, it can be reached only by objects within the package or which inherit from it. We will uses the getIn(), getOut(), and getErr() methods in testing.

The print() methods is overloaded. It can tell whether it is passed a Friend, Item or Loan. However, the read methods cannot be overloaded because the return value is not part of the method signature that distinguishes between methods: `void print(Friend)` is different from `void print(Item)`, but `Friend read()` is not different from `Item read()`. We need to distinguish them by their names (i.e., readFriend() and readItem()). Similarly, print(ArrayList<Friend>) is not distinguished from print(ArrayList<Item>) because they are both printing ArrayLists. We could solve this problem by defining a class called Friends and Items, which could be distinguished in the method signature, but instead we decided to distinguish the methods in their names.

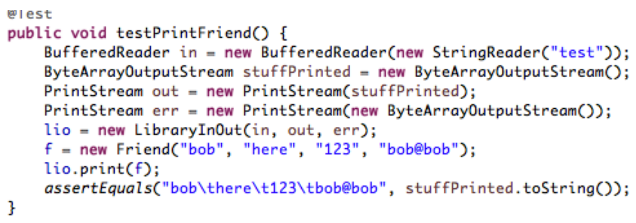
To build Build JUnit tests for the class, we create a class that uses string input and output streams. These streams print output into a string and read input from a string. A JUnit test that creates a new LibraryInOut class with streams that read and write to strings is shown in figure 2.



```
@Test
public void testConstructor() {
    BufferedReader in = new BufferedReader(new StringReader("test"));
    PrintStream out = new PrintStream(new ByteArrayOutputStream());
    PrintStream err = new PrintStream(new ByteArrayOutputStream());
    lio = new LibraryInOut(in, out, err);
    assertThat(lio.getIn(), instanceOf(BufferedReader.class));
    assertThat(lio.getOut(), instanceOf(PrintStream.class));
    assertThat(lio.getErr(), instanceOf(PrintStream.class));
    lio = new LibraryInOut(in);
}
```

Figure 2: Construction a LibraryInOut class with string input and output

The line `BufferedReader in = new BufferedReader(new StringReader("test"))` creates a new stream that will produce the characters 't', 'e', 's', and 't'. when it is read. The line `PrintStream out = new PrintStream(new ByteArrayOutputStream());` produces a stream that will store all of the output that is printed to it in a string. Calling `toString()` on the `ByteArrayOutputStream` returns the string that contains all of the things that were printed to it. You can see an example of a test that uses this strategy in figure 3.

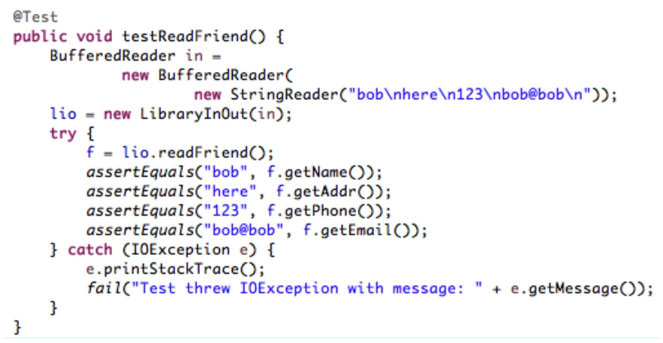


```
@Test
public void testPrintFriend() {
    BufferedReader in = new BufferedReader(new StringReader("test"));
    ByteArrayOutputStream stuffPrinted = new ByteArrayOutputStream();
    PrintStream out = new PrintStream(stuffPrinted);
    PrintStream err = new PrintStream(new ByteArrayOutputStream());
    lio = new LibraryInOut(in, out, err);
    f = new Friend("bob", "here", "123", "bob@bob");
    lio.print(f);
    assertEquals("bob\there\t123\tbob@bob", stuffPrinted.toString());
}
```

Figure 3: JUnit print test that uses string buffers

The `ByteArrayOutput` stream needs to be names, so we can access the string after the test has run. We can see that when the test succeeds, `lio.print(f)` will have printed each of the strings in the Friends' fields separated by tab characters.

To test reading, we need to supply the functions to be tested with input. Figure 4 shows an example of a read test.



```
@Test
public void testReadFriend() {
    BufferedReader in =
            new BufferedReader(
                    new StringReader("bob\nhere\n123\nbob@bob\n"));
    lio = new LibraryInOut(in);
    try {
        f = lio.readFriend();
        assertEquals("bob", f.getName());
        assertEquals("here", f.getAddr());
        assertEquals("123", f.getPhone());
        assertEquals("bob@bob", f.getEmail());
    } catch (IOException e) {
        e.printStackTrace();
        fail("Test threw IOException with message: " + e.getMessage());
    }
}
```

Figure 4: JUnit read test that uses string buffers

Here we see hat the StringReader is initialized with a string in which all of the fields in a Friend class are separated by values. We can read from the string into a Friend class and then check that the fields have been set correctly. The function read() throws an IOException so the function that calls it will throw an IOException. We need to catch the exception. If it throws the exception, the test fails.

Next Week's Demo Marks (50 Marks)

Stories 14 – 24 (5 Marks each)

- JUnit Test (3 Marks)
- LibraryInOut method (2 Marks)