Overview

Your code from Part 3 should successfully read in data from various files including **`items_and_user_data.txt`** into your model. We are now going to add more functionality to the model.

So, in this part of the project, you will

- Change the way that we store our data;
- introduce a new class that allows a library user to borrow an item for a specified period of time. To assist this, we will make use of
    - a **`DateUtil`** class that provides functionality for dealing with dates and the **`Date`** class of **`java.util`**;
    - a **`Diary`** class that provides useful features for storing borrowings.

Again, read ALL of each step before you start to code

Step 1 = lots to read but mostly explanation 😊

In this step we will change the way that we store our data. When you do make these changes, do not delete the old declarations and any other "original" code – simply comment these out – so that we can allocate marks to this code (as well as to your new code).

The library users and library items are currently stored in two **`ArrayList`** objects, one for the items and a second for the library users. In Slides 12, we saw how an **`ArrayList`** implements the **`List`** interface. This means that your two **`ArrayList`**objects can also be regarded as being of type **`List`**. We saw a somewhat similar situation in Chapter 10 with the **`Network`** project where **`MessagePost`** and **`PhotoPost`** objects could also be regarded as being of type **`Post`**.

To check this, make appropriate changes to the ***declarations*** of the **`ArrayList`** fields in the **`Library`** class so that they are now of type **`List`**. Check that your code still compiles and executes correctly. Nothing has really changed as the two lists still reference array list objects ☺

An interface such as **`List`** allows a type of multiple inheritance, as discussed in Chapter 10, but it is important to remember that the **`add()`** method is ***not*** inherited by the classes **`ArrayList`** and **`LinkedList`**. Instead, the fact that the classes ***implement*** the interface implies that they ***must*** each provide an **`add()`** method consistent with the one specified by **`List`**. Although the **`ArrayList`** and **`LinkedList`** classes all implement the **`List`** interface, they have different ways of storing the elements of a list and each has their own advantages and disadvantages.

In our model, all instances of the **`LibraryUser class`** have a field **`userID`** that is intended to uniquely identify each library user. Similarly, all **`LibraryItem`** objects should have a field **`itemCode`** that uniquely identifies them. If we are going to have these unique identifiers, it makes sense to re-think the way that we store library users and items. Recall Slides 6.23-27 that gave examples of key/value pairs and briefly discussed mappings and the **`HashMap`** class. This class is also an example of a class that implements an interface, in this case the **`Map`** interface.

We can regard the **`Map`** interface as playing the same role for "mappings" as the **`List`** interface plays for "lists" and any class that implements it will be able to store key/value pairs. A very significant feature of a mapping is that the value corresponding to a given key can be easily retrieved, unlike retrieving objects from a list where it is necessary to search through the list to find the object you want.

One advantage of using **`Map`**s instead of **`List`**s is that it will be a simple matter to easily retrieve a **`LibraryUser`** object corresponding to a given **`userID`** or a **`LibraryItem`** object corresponding to a given **`itemCode`**. Hence, we will now make the change to the way that we store the library users and item and use a **`Map`** instead of a **`List`**.

Keeping your declarations for `List`, **start coding** by adding two new variables `customerMap` and `itemsMap`. Re-read Slides 6.23-27, and think carefully about your choice of key value. These new declarations more accurately reflect, in the code, how the customers and items are now stored but users of the system can still think of having a "list of customers" and a "list of items".

So, just as we needed earlier, a particular type of `List` (e.g. `ArrayList` or `LinkedList`) in which to store the library users and items, we now need a particular type of `Map`. We have seen a `HashMap` used in the Tech Support System project of Chapter 6 so we will choose to use a `HashMap` at this point. If we decide later that it is not the best choice then, if you have coded sensibly, it is not difficult to change to another kind of `Map` (in a similar way to changing the kind of `List` earlier in this step).

Next, **in addition to** the two existing calls to the constructor of the `ArrayList` class (which you should now comment out), add two new calls to an appropriate constructor of a different class.

Unfortunately, even if you perform these changes correctly, it is not possible to check your work (at the moment) since you will find that your code will no longer compile.

So, you will need to go through your code making necessary modifications**. No major changes are needed, simply changes to "details"** e.g. unlike a `List`, a `Map` does not have an `add()` method so you will need to replace statements such as

```
itemList.add(libraryItem);
```

by some appropriate alternative. However, you will probably find it useful know that the `values()` method of a `Map` object returns a "collection view" (e.g. a `List`) of the objects stored in a `Map`.
When you can successfully compile the project, test that the code executes correctly. You should be able to read in data, display library items and library users, write out user data in much the same way as you could at the end of Part 3 of the project.

Step 2

The purpose of this step is to explore the `Date` and `DateUtil` classes. The `DateUtil` class can now be downloaded from BlackBoard and added to your project. This class imports the `Date` class which is in the standard Java libraries.

An instance of the `Date` class "represents a specific instant in time, with millisecond precision" (see html documentation for the class) and can be used, together with classes such as `Calendar`, `GregorianCalendar` and `SimpleDateFormat`, to represent and manipulate dates and times. In this project, we will concentrate on using it to represent simply "dates" (e.g. **28/02/2024**) rather than "dates and times" (e.g. **28/02/2024 14:59**).

Open the project into which you downloaded the class `DateUtil` and read the introductory javadoc comments. Note the following:

- The first two methods `convertDateToLongString()` and `convertDateToShortString()` convert `Date` objects to easily understandable strings e.g. `"Wednesday, 28 February 2024"` and `"28-02-2024"` where the "style" of the "long" and "short" versions of a date is determined by the fields `longDatePattern` and `shortDatePattern`.
    - ***Do not change these patterns*** because `Library` expects dates in these styles ! However it is clear that alternative "styles" are possible and the class could easily be enhanced by adding `setlongDatePattern()` and `setshortDatePattern()` methods.
- The `convertStringToDate()` method converts a string, in the style `"28-02-2024"`, to a corresponding `Date` object. We do this because such dates can easily be "manipulated" e.g. the following code

```
Date today = DateUtil.convertStringToDate("28-02-2024");
Date todayWeek = DateUtil. incrementDate(today, 7);
System.out.println(DateUtil.convertDateToLongString(todayWeek));
```

  will output `"Wednesday, 6 March 2024"` to the terminal window.

Give yourself confidence in using this class by looking at all the methods and reading their documentation. Once you have done this, (assuming you have added this class to your project), write code in your Test class that calls the method `convertStringToDate()` twice and then, using your two `Date` objects, calls the `daysBetween()` method and prints out the number of days between the two dates that you have chosen.

We shall now use these classes to allow library users to borrow an item.

Step 3

A user can already use the library's **`printAllItems()`** method to see the items in the library and, having seen them, that user may then wish to borrow a particular item such as a DVD or a specialist book. Such a reservation will need to specify the library user, the item, the start date for the reservation and the number of days the item is to be borrowed for. It will also be convenient to give each reservation a unique id -- in this case we will store a six digit numerical string which will be generated sequentially e.g. **`000001`**, **`000002`**, **`000003`**, **`000004`**, etc

So, you should now introduce an LibraryReservation class with:

- three fields **`reservationNo`**, **`itemCode`**, **`userID`**, all of type **`String`**;
- a field **`startDate`** of type **`Date`**: the date of the first day of the reservation;
- a field **`noOfDays`**: the duration of the reservation.

Next write the code for the constructor for the **`LibraryReservation`** class which should have five parameters **`reservationNo`**, **`itemCode`**, **`userID`**, **`startDate`** and **`noOfDays`**, in the order specified here. The type of each parameter should be the same as the corresponding field ***except*** for the **`startDate`** parameter which should be of type **`String`**. The argument passed for the **`startDate`** parameter should be in the **`shortDatePattern`** style mentioned above e.g. **`"28-02-2024"`**.

Any reservations that are made will need to be stored in the library so next you need to modify the **`Library`** class to handle the reservations by following these steps:

- Introduce a **`libraryReservationMap`** field to store the reservations.
- Write a simple **`storeLibraryReservation()`** method similar to the **`storeItem()`** method (***not*** the **`storeUser()`** method which is more complicated). You should assume that the **`LibraryReservation`** object passed to the method will have a unique **`reservationNo`**.
- Write a method **`generateReservationNo()`** which generates the unique reservation number for a reservation. You should also "pad out" your numbers with zeros to produce, for example, **`000001`**, **`000002`**, **`000003`**, **`000004`**, etc.
    - If we close our model then we should save our last generated number to a file so that we could then read it back in when we reopened the model. However, let's leave that for now. ☺
    - If this step proves too difficult, simply generate any random number & move on to the next bullet point.
- Write a **`getLibraryReservation()`** method.
- Add a method with the following header

```
public void makeLibraryReservation(String userID, String itemCode,
                                   String startDate, int noOfDays)
```

  Initially the method should simply
    - get a reservation number for the reservation;
    - create an **`LibraryReservation`** object;
    - add it to the list of reservations.

  However, it is obvious that not all attempts to make a reservation will be valid -- for example, there may not be an item corresponding to the id passed to the method. So make the method return a **`boolean`**, rather than **`void`**, and add checks at the beginning of the method to make sure that the four parameters are passed valid values:
    - if they are valid, the method should return **`true`**;
    - otherwise the method should output a message to the terminal window and return **`false`** without creating an **`LibraryReservation`** object, etc.

  There is also the possibility that a particular item may have already been reserved for all or part of the reservation period. ***Do not attempt to check for this situation, we will return to this issue later ☺.***

- Add methods **`printDetails()`** and **`printLibraryReservations()`** to the **`LibraryReservation`** and **`Library`** classes respectively. These methods allow you to check that a "valid" reservation has been correctly added to the reservation list.
- Finally for this step, add methods **`writeLibraryReservationData()`** and **`readLibraryReservationData()`** to the **`Library`** class. These methods should be similar to the **`writeUserData()`** method and will need **`writeData()`** and **`readData()`** methods in the **`LibraryReservation`** class. You should be able to copy and paste much of the code ☺ but note that you will need to use the **`DateUtil`** class to convert **`Date`** objects to and from **`String`** objects.

The following step of the project is not needed for later steps of the project so it may skipped if you find it too difficult though marks are allocated to this step.

Step 4

If reservations were for one day only then it would be possible to easily check to see if an item selected by the library user was free on a particular day. But reservations will typically span a number of days and that will make the checking more difficult e.g. the library item that a user might want to hire may be available on some days but not on others. To overcome this problem and to provide other features, we have written a class **`Diary`** that can be downloaded from BlackBoard and added to the project. A **`Diary`** object, as its name suggests, represents a diary which stores the reservations that have been made and the class provides the following public methods:

```
public void addLibraryReservation(LibraryReservation reservation)
public void printEntries(Date startDate, Date endDate)
public void deleteLibraryReservation(LibraryReservation reservation)
public LibraryReservation[] getLibraryReservations(Date date)
```

Open the class and examine the Javadoc documentation which gives a brief description of each of these methods. If you scroll down the code, you will find at line 120, that the class contains an ***inner class***, called **`DayInDiary`**:

- Just as a class can have fields, constructors and methods, it can also contain inner classes.
- It would have been possible to make **`DayInDiary`** a public class in the usual way but sometimes there are advantages in having an inner class:
    - in this case, the class is introduced only to help produce the functionality of the **`Diary`** class, it is never accessed (directly) by, for example, the **`Library`** class;
    - making it an inner class makes it **`private`** (it is not possible to have an inner **`public`** class) and so improves ***encapsulation***, see Chapter 8.
- The class **`DayInDiary`** itself contains yet another inner class, **`Entry`** ! However, note that the code could have been written slightly differently by making **`Entry`** an inner class of **`Diary`** rather than of **`DayInDiary`**. But in its present form, it again improves encapsulation.
- Although the inner classes are private and have fields, constructors and methods declared to be **`private`**, the **`private`** modifier does not affect accessibility in the usual way for inner classes. Fields and methods are freely accessible between the outer and inner classes: e.g. an inner class can access a private field in **`Diary`** and vice-versa !

Note: you should ***NOT*** make any changes to the **`Diary`** class.

To make use of this class, add a field **`diary`** to **`Library`**, initialise it in the constructor and add a call to the **`Diary`** class's **`addLibraryReservation()`** method in the **`storeLibraryReservation()`** method. Whenever you now "make a reservation" using the **`makeLibraryReservation()`** method, the reservation will also be added to the "diary". However, remember that we are not storing the reservation objects "twice" and using, potentially, a lot of memory: both the **`LibraryReservationMap`** and **`diary`** objects are merely storing the addresses of (or "pointers to") the same **`LibraryReservation`** objects.

Compile and depending on whether you have been continuing with your accessor methods then your code should compile.

Now write a new method for the **`Library`** class, **`printDiaryEntries()`** with **`String`** parameters, that calls the **`printEntries()`** method of the **`Diary`** class. This will enable a user of the system to display diary entries for a specified period. You might find it convenient to add a **`toString()`** method to the **`LibraryReservation`** class

that returns a string combining the **`reservationNo`**, **`userId`** and **`itemID`** in an easily readable form. Note that, for the purposes of this project, a reservation for a one day period (i.e. **`noOfDays`** is equal to **`1`**) will have the same start and end date.

Finally, for this step

- amend the **`makeLibraryReservation()`** method so that it is not possible to make the reservation if the item is already reserved for all or part of the reservation period. **Hint:** think how you would do this in real life and then look at the **`Diary`** class for any method that might help you.
- write a method **`deleteLibraryReservation()`** that is passed a reservation number and deletes the corresponding reservation from the reservation system.

Hand in details:

- You will need to upload **TEN separate** .java files (**Library.java**, **Book.java**, **Periodical.java**, **CD.java, DVD.java, PrintedItem.java, AudioVisual.java, LibraryItem.java, LibraryUser.java** and **LibraryReservation.java**) - to **separate** upload areas on BB**.**
- **If you upload your code incorrectly then we will have no option but to allocate zero to that part of your work.**
- **The deadline to do this is 16.00 Friday 22nd March 2024**

Note that by the act of following these instructions and handing your work in, it is deemed that you have read and understand the rules on plagiarism as written in your student handbook.