M3 Individual Android

Submit Assignment

Due Monday by 11:59pm **Points** 50 **Submitting** a file upload
Available Jan 10 at 12am - Feb 5 at 1:59am 26 days

M3 - Android and MVVM

Purpose

This exercise will ensure you can build and execute an Android application on your computer. You will also add functionality across the application stack to solidify your understanding of Model-View-ViewModel.

Recall from class the following guidelines:

Model - the classes that hold data for the application (Entities), and the business rules (interactors). These are normally POJO (plain old java objects) and are independent of any development frameworks you are using. In fact, as much as possible, we want to avoid making these classes dependent on specific frameworks, databases, network protocols, etc. The actual POJOs are found in the entity package, while the business rules are in the model package.

View - the screens that display and get information from the user. (in android, these are the .xml layout files and the Activities)

ViewModel - a class associated with the view that has all the event handling code and code that interacts with the model to update information obtained from the model. In Android, these are subclasses of the new ViewModel component. A true view model would be responsible for data binding between the view and the model. In Android, this would require us to learn about the new LiveData classes. We will not be using LiveData in this lab however.

Running the App

You should download the M32MVVM.zip file.

Lab3MVVM.zip

Extract the file into a project directory. If using Android Studio, I have included my idea files so you can import the project. If using a different IDE, or coding from the command line, simply import the code files. Note that you probably will need to update some settings like gradle and sdk location. These can be done from Preferences.

If you examine the project view, you will see there are packages for entity, model, viewmodel and views. In Android, our views are the Activities and supporting UI code in the views package and the layout files located in res/layout/

If you run the project, it should build and execute normally. You should be able to see the list of courses on the starting screen. Clicking the red circle will allow you to add a new course. Swiping a course off to one side allows you to delete that course.

If you click (touch) one of the courses, the screen will change to a list of the students in the class. On this screen, you can edit students. Adding new students has not been implemented yet since it involves not just adding a student to the list of all students, but also registering them for the current course.

If you go back to the main screen, you should see the three dots in the upper right hand corner of the application. Click on them and then select View All Students. This screen is where you want to do all your work. Clicking on an individual student allows you to edit that student's details and updates the model with the new information. Clicking the red + sign, will add a new student to the model (but not register them for any classes). Newly added students should show up automatically in the View All Students window. (As stated previously, accessing the students from an individual course and adding from there is NOT implemented). Work through the View All Students window!

New Features

Your customer likes the application so far, but they want one minor change. They want to include the student's class standing (freshman, sophomore, etc.). This will possibly require a change in each component (Model, View and ViewModel) of the MVVM stack.

Normally when maintaining code and making feature enhancements and bug fixes, you can start wherever you like in the stack. I usually like to start with the Model.

Model changes

Since the customer's request is to add information to the Student, we can focus on the Student class in the Entity package.

The next step uses enum. If you are unfamiliar with enum and when/why it is used, review any basic Java text or online source.

1. To support the class standing, we will first need to create a new class named ClassStanding which is an enum with the values FRESHMAN, SOPHOMORE, JUNIOR, and SENIOR. Give each class a two character representation "FR", "SO", "JR" and "SR". These two letter codes will be attributes of the enum values.
2. Now add an attribute to Student which holds the ClassStanding enum. NOTE: ClassStanding is NOT a String, it is one of the enum values like: ClassStanding.FRESHMAN.
3. Add a getter and setter for the ClassStanding.
4. Make a new 3 param constructor which includes name, major and class standing. Keep the old two param constructor for backwards compatibility and use constructor chaining to give a default class standing of Freshman. Remember that we chain to the constructor with the MOST parameters, so you may have to move some code out of the original 2 parameter constructor.
5. Change the toString method to contain the class standing also.
6. In the Model package, there is a class called Repository. There is a method called updateStudent. You will need to add a line of code there to update the ClassStanding along with the name and major.

In a real project we would also be updating our JUnit tests along with making these changes so we could verify that the code is working correctly. Unfortunately, we have not learned JUnit, so we can move forward to the next steps.

View changes

Now you will need to add some widgets to the view so that when adding a new student, you can get the class standing from the user. Open the content_edit_student.xml file, located in the "res" section under layout.

If you click the Design tab, you can see the current screen layout. Click on the text tab and you can view the .xml code. You can either hand edit the .xml or you can do drag and drop design from the Design view.

You will see that the dialog is laid out in a Relative layout [see Appendix] with labels and input fields. You will need to add to the layout some way to input the class standing. I used a spinner (ComboBox/dropdown) in my solution, but 4 radio buttons would work. You could use a text field and type in the standing, but that is error prone and you would need to validate that an incorrect entry was not made.

If you want to use a spinner, see the Appendix for a bit more info on how a spinner works.

Now you should be able to run the app again, just to make sure everything is ok. You should be able to tap on a student to edit and your dialog should pop up with your new UI features showing. It won't actually do anything yet, and if you used a combobox like me, it won't have any values in it. To make the display actually work, we have to edit one more class, the EditStudentActivity. The Activity is the Java code part of the View that supports the .xml file.

Go to EditStudentActivity.java in the Views package. You will need to add attributes for the widget(s) that you added into the .xml file. In the onCreate method, you will need to use findViewById to attach to the correct widget. Also in onCreate, you will need to set up your widget if necessary. For example, it is there that you will populate a spinner if you chose to use one.

If the intent [see Appendix] to start this activity included a Student object, then a student is being edited. When editing a student, you should ensure the data shown is that student's data. This is already done for name, id and major. You need to do this for class standing. To put it explicitly, when you tap on a Junior to edit that student, Junior should already be selected in your widget of choice. You will also need to add a line to the onAddPressed method.

Now we will need to edit the recycler view [see Appendix], so that class standing is displayed along with the major. How you do this is up to you, but will likely involve editing StudentAdapter.java and student_item.xml. In my solution, I (Bob) edited student_item.xml to add new TextView so that the major is displayed in the cardview. Look at the course item to get an idea of how to add it to the major the same way the school and number are shown in the course. Another option that I (Arshad) used is to utilize the major text field and include the class standing there.

You may want to see the Appendix for a quick note on the ViewHolder pattern, which is used here.

Now run the app. You should be able to edit students and their standing and see it reflected in the UI. If not, then check your code through the complete stack. Are you calling the setters in the model to update the information? Are you pulling the information out of the widget correctly?

NOTE: If using a spinner, it doesn't matter if the entries are of the form "FRESHMAN" or "FR". Either will be accepted. It is easier to hook the spinner up to values() result rather that make an array of strings yourself.

ALL DONE!

Congratulations you have edited a full stack MVVM app. You also caught a glimpse of the life of a maintenance programmer. Many of you will not work on new applications, you will work on established products to add new features, just like we have done here.

It is not necessary to implement registration or any functionality on the course details screen. Just add this one piece of information to the student.

Submission

Zip up your complete project directory and submit to the assignment. On Windows, you can do this by going to File>Export project to zip. There should be a similar option for Mac.

Appendix

Android development has a lot of terms, and I wanted to explain some of them here.

Intent:

An intent is the way Android switches between screens. The general process is that you instantiate a new Intent object with the source and destination screens, add any "Extras" (info the new screen needs) and then start the new Activity.

RecyclerView:

This is the best way of displaying a list in Android. It is a strict successor to the older and slower ListView. To populate a RecyclerView (or any list-like widget) you need an **Adapter**. This "adapts" objects of whatever type into the specific sets of data needed for the RecyclerView .

ViewHolder Pattern:

This is a popular way of writing an Adapter. It is a bit complex, but the important part is that each entry in a RecyclerView is a "**view**". Each entry involves constructing the view in a method called onCreateViewHolder. This tells the view what its layout will be (defined by an xml). Then, a specific object (for example, a specific student) is "**bound**" to the view in onBindViewHolder. This step involves actually taking the data from the object and putting it into the right fields in the view. For a student, this would involve putting their name and major into the text fields of the view.

Spinner:

You can think of a spinner as a simple list: it will use an adapter just as a RecyclerView. Because a spinner is so simple, the ArrayAdapter class is used instead of writing your own adapter. It simply takes in a list of objects you want to display. One of the benefits of enums is that they provide a convinient way to access a list of all their values in the static .values() method.

Another concern for a spinner is how to set the default selection. To do this, you need to provide an int representing the value's index. Enums have another convinient method here: the instance method .ordinal().

Relative Layout:

This is one way to do things along the lines of "put the spinner below the textview" or "put the button at the bottom of the screen". As the name suggests, this is a way of laying out widgets relative to other widgets or the borders of the screen. It uses xml tags such as "**layout_below**" to set the widget below another or "**layout_alignParentStart**" to tell the widget to be as far right (on English devices) as possible. For example, when you add your widget, you may want to use tags like those to put your widget below the major spinner.

M3 Individual Android

Criteria	Ratings		Pts
enum for ClassStanding created	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
Class standing attribute added in Student class	**4.0 pts** **Full Marks**	**0.0 pts** **No Marks**	4.0 pts
getters and setters added for ClassStanding	**3.0 pts** **Full Marks**	**0.0 pts** **No Marks**	3.0 pts
New constructor and constructor chaining done correctly	**4.0 pts** **Full Marks**	**0.0 pts** **No Marks**	4.0 pts
toString method implemented correctly	**3.0 pts** **Full Marks**	**0.0 pts** **No Marks**	3.0 pts
Input widget for class standing added to dialog	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
Student list displays correct information including new data	**8.0 pts** **Full Marks**	**0.0 pts** **No Marks**	8.0 pts
Newly created student is added to the list	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
Default information for student is correctly displayed in the edit dialog	**8.0 pts** **Full Marks**	**0.0 pts** **No Marks**	8.0 pts
View Model class updated correctly	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
			Total Points: 50.0