School of Information Technology

Course:	Diploma in Information Technology
Module:	IT2161 Mobile Applications Development

Practical 10: View Model and Room

Objectives:

- Understanding of View Model and Room
- Creating an application that uses View Model and Room to persist app data.

Exercise 10: ViewModel & Room

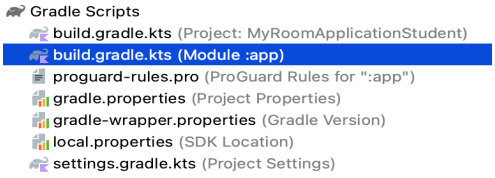
Notes

- You should use the project starter file for this exercise.

Steps

1. Download the project starter file into a directory.
2. Start Android Studio and open up the Android project that you downloaded.

Exercise 10.1: Set up Gradle [build.gradle(Module)]



3. Add the kapt annotation processor Kotlin plugin.

```
//TODO 1 : Apply the kapt annotation processor Kotlin plugin
id("kotlin-kapt")
```

4. Some of the APIs will require 1.8 jvmTarget, so add that to the android block.

```
//TODO 2: Add the 1.8 jvmTarget to the Android block
kotlinOptions {
    jvmTarget = "1.8"
}
```

5. Add the necessary dependencies for Room and View Model.

```
dependencies { this: DependencyHandlerScope

    implementation("androidx.core:core-ktx:1.9.0")
    implementation("androidx.appcompat:appcompat:1.6.1")
    implementation("com.google.android.material:material:1.10.0")
    implementation("androidx.constraintlayout:constraintlayout:2.1.4")

    //TODO 3: Add the necessary dependencies for Room and View Model
    // Room components
    val room_version = "2.6.1"
    val lifecycleVersion = "2.6.0"

    implementation("androidx.room:room-runtime:$room_version")
    annotationProcessor("androidx.room:room-compiler:$room_version")

    // To use Kotlin annotation processing tool (kapt)
    kapt("androidx.room:room-compiler:$room_version")
    // optional - Kotlin Extensions and Coroutines support for Room
    implementation("androidx.room:room-ktx:$room_version")
// Lifecycle components
    implementation ("androidx.lifecycle:lifecycle-viewmodel-ktx:$lifecycleVersion")
    implementation ("androidx.lifecycle:lifecycle-livedata-ktx:$lifecycleVersion")
    implementation ("androidx.lifecycle:lifecycle-common-java8:$lifecycleVersion")
    implementation ("androidx.activity:activity-ktx:1.7.0")
```

Exercise 10.2: Entity class [Contacts]

6. Open up Contacts and add in the notation to include the information of the table and column information.

```
//TODO 4:
// - Update the entity class with annotations
@Entity(tableName = "contacts_table")
data class Contacts(@PrimaryKey(autoGenerate = true) @ColumnInfo(name = "id") val id : Int,

                    @ColumnInfo(name="name")val name: String,

                    @ColumnInfo(name="num")val num : String)
```

Exercise 10.3: Data Access Object(DAO) class [ContactsDao]

7. Complete the interface by adding in the function prototypes for
   - Retrieval of all contacts
   - Insertion of a contact
   - Deletion of a contact

```
interface ContactsDao{

    fun retrieveAllContacts() : List<Contacts>


    fun insert(newContacts : Contacts)


    fun delete(delContact : Contacts)
}
```

8. Add in the Dao notation for queries, insert and delete.

```
@Dao
interface ContactsDao{
    @Query( value: "Select * from contacts_table")
    fun retrieveAllContacts() : List<Contacts>

    @Insert(onConflict = OnConflictStrategy.ABORT)
    fun insert(newContacts : Contacts)

    @Delete
    fun delete(delContact : Contacts)
}
```

9. To observe data changes, use “Flow” from kotlinx-coroutines

```
interface ContactsDao{
    @Query( value: "Select * from contacts_table")
    fun retrieveAllContacts() : Flow<List<Contacts>>
```

Exercise 10.4: RoomDatabase [ContactsRoomDatabase]

10. Update ContactsRoomDatabase to an abstract class that implements RoomDatabase.
11. Add in the annotation for the class to be a room database and use the annotation parameters to declare the entities that belong in the database and set the version number.
12. Expose DOA through an abstract getter method for @Dao

```
//  - Define a singleton to prevent having multiple instances of the database open
@Database(entities = arrayOf(Contacts::class), version = 1, exportSchema = false)
abstract class ContactsRoomDatabase : RoomDatabase() {

    abstract fun contactsDao(): ContactsDao

    companion object {

    }
}
```

13. Define a singleton to prevent having multiple instances of the database opened at the same time.

    getDatabase returns the singleton. It will create the database the first time it is accessed, using Room's database builder to create a RoomDatabase object in the application context from the ContactsRoomDatabase class and names it "contacts_database".

```
abstract class ContactsRoomDatabase : RoomDatabase() {
    abstract fun contactsDao(): ContactsDao
    companion object {
        @Volatile
        private var INSTANCE: ContactsRoomDatabase? = null
        fun getDatabase(context: Context, scope : CoroutineScope) : ContactsRoomDatabase {
            return INSTANCE ?: synchronized( lock: this){

                val instance = Room.databaseBuilder(context,
                    ContactsRoomDatabase::class.java, name: "contacts_database"
                    ).build()
                INSTANCE = instance
                instance  ^synchronized
            }
        }
    }
```

Exercise 10.5: Repository [ContactsRepository]

14. Update the repository class to hold a list of contacts from the DAO.

```
class ContactsRepository(private val contactsDao: ContactsDao) {
    //TODO 8:
//  - Update the repository class to hold a list of contacts from DAO.
    val allContacts = contactsDao.retrieveAllContacts()
```

15. Add in the necessary functions to do insert and delete of contacts using the DAO

```
//TODO 9:
//  - Add in the necessary functions to do insert and delete of contacts using the DAO
    suspend fun insert(contact : Contacts){
        contactsDao.insert(contact)
    }

    suspend fun delete(contact : Contacts){
        contactsDao.delete(contact)
    }
```

Exercise 10.6: ViewModel [ContactsViewModel]

16. Update ContactsViewModel to implement ViewModel.
17. Create a LiveData list to cache what allContacts in repository returns.

```
//TODO 10:
//  - Update ContactsViewModel to implement ViewModel
class ContactsViewModel(val repo: ContactsRepository) : ViewModel() {
    //TODO 11:
    // Create a list of contacts
    // - Use LiveData to cache all contacts
    val allContacts: LiveData<List<Contacts>> = repo.allContacts.asLiveData()
```

18. Make use of coroutines to insert and remove contacts.

```
//TODO 12 :
// - Launch a new coroutine to insert the contact
fun insert(contactItem : Contacts) = viewModelScope.launch(Dispatchers.IO){ this: CoroutineScope
    repo.insert(contactItem)
}
//TODO 13 :
// - Launch a new coroutine to remove the contact
fun remove(contactItem : Contacts) = viewModelScope.launch(Dispatchers.IO){ this: CoroutineScope
    repo.delete(contactItem)
}
```

19. Implement ContactsViewModelFactory to create ContactsViewModel using the ContactsRepository.

```
//TODO 14:
// - Implement ContactsViewModelFactory to create ContactsViewModel using the ContactsRepository

class ContactsViewModelFactory(private val repo : ContactsRepository):ViewModelProvider.Factory{

    override fun <T : ViewModel?> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom(ContactsViewModel::class.java)) {


            return ContactsViewModel(repo) as T
        }

        throw IllegalArgumentException("Unknown viewmodel class")
    }

}
```

Exercise 10.7: Application [MyContacts]

20. Declare a coroutine scope that is to be used later when creating a database instance.
21. Create a database and repository instance using a lazy delegation.

```
class MyContacts() : Application(){
//TODO 15:
// - Create a database and repo instance
// - Make use of the "by lazy" delegation to state that the objects are the cru
    val appScope = CoroutineScope(SupervisorJob())
    val db by lazy {ContactsRoomDatabase.getDatabase( context: this,appScope)}
    val repo by lazy{ContactsRepository(db!!.contactsDao())}
}
```

Exercise 10.8: Activity [MainActivity]

22. Declare and initialize a ContactsViewModel variable.

```
//TODO 16:
//  - Create the ViewModel
private val contactsViewModel: ContactsViewModel by viewModels() {

    ContactsViewModelFactory((application as MyContacts).repo)
}
```

23. In onCreate, add an observer for the allContacts live data from ContactViewModel.

    On any changes, update the array adapter with the new item list.

```
//TODO 17:
//  - Add an observer for the allContacts LiveData from ContactViewModel

contactsViewModel.allContacts.observe( owner: this, Observer {  it: List<Contacts>!

    var contactsConvertList = mutableListOf<String>()
    allContacts = it
    for(contact in it){
        contactsConvertList.add("${contact.name} - ${contact.num}")
    }
    it?.let{  it: List<Contacts>
        contactsAdapter = ArrayAdapter( context: this,
            android.R.layout.simple_list_item_1, contactsConvertList)
        contactsLV.adapter = contactsAdapter
    }
    toggleVisibility()
})
```

24. In onContextItemSelected, remove the selected contact using ViewModel.

```
override fun onContextItemSelected(item: MenuItem): Boolean {

    //TODO 19:
    //  - Make use of view model to remove the selected contact.
    val info = item.menuInfo as AdapterView.AdapterContextMenuInfo

    if(allContacts!=null) {

        var contact = allContacts!!.get(info.position)
        contactsViewModel.remove(contact)

        toggleVisibility()
    }
    return super.onContextItemSelected(item)
}
```

25. Implement onActivity result to insert the new contact information using ViewModel.

```
//TODO 18 :
// - Implement onActivityResult to insert the new contact using the ViewModel after

override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
    super.onActivityResult(requestCode, resultCode, data)
    when(requestCode){
        1->{

            if(data!= null) {
                var name = data!!.getStringExtra( name: "name")

                var num = data!!.getStringExtra( name: "num")

                contactsViewModel.insert(Contacts( id: 0,name,num))
            }
        }
    }
}
```

Exercise 10.8: Unguided

- Add in two properties into contacts. Address and Notes
- Add in the necessary codes to allow users to edit and update properties