**Week 9 (Room Database)**

**Part 1: Create and configure the project**

1. Create an Empty Views Activity project. Name the project as **WordDatabase**. Remove unnecessary classes and layout files (refer to your tutor or teaching staff).
2. Add the following code to your build.gradle (Project: app) file.

ext **{** lifecycle\_version = '2.6.1'  
 room\_version = '2.5.1'  
**}**

1. Add the following code to your build.gradle (Module: app) file. Sync the project.

plugins **{** id 'com.android.application'  
 id 'org.jetbrains.kotlin.android'  
 **id 'kotlin-kapt'**  
**}**

android **{**

**...**

buildFeatures **{** viewBinding = true  
**}**

**}**

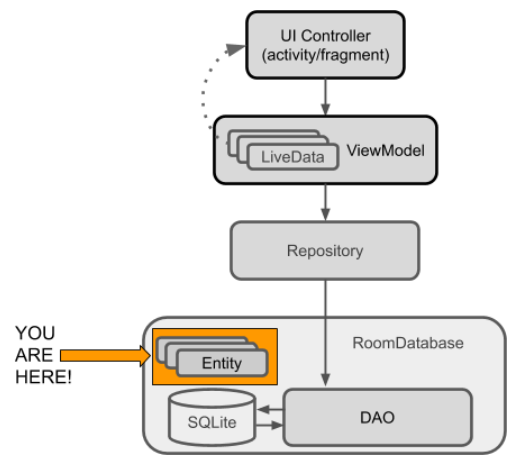
dependencies **{  
  
 ...**  
 ***// Room libraries* implementation "androidx.room:room-runtime:$room\_version"  
 kapt "androidx.room:room-compiler:$room\_version"  
 implementation "androidx.room:room-ktx:$room\_version"**

***// Coroutines***

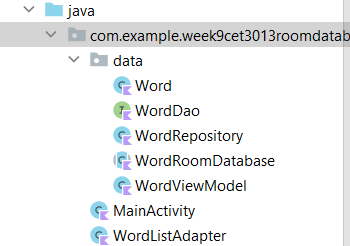
**implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.4'  
 implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.6.4'**

**}**

**Part 2: Create the Word entity**



1. Create a package called **data**. We have to place all the Room classes in this package.



1. Create a class called Word.

public class Word(var mWord:String) {  
  
}

1. Update your Word class with annotations, as shown in the code below.

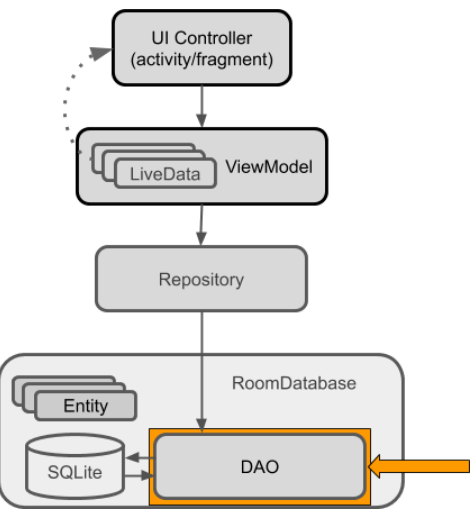
@Entity(tableName = "word\_table")  
public class Word(  
 @PrimaryKey  
 @NonNull  
 @ColumnInfo(name = "word") var mWord:String) {  
  
}

1. If you get errors for the annotations, you can import them manually, as follows:

import androidx.annotation.NonNull  
import androidx.room.ColumnInfo  
import androidx.room.Entity  
import androidx.room.PrimaryKey

1. **Tip on auto-generating keys:** To [auto-generate](https://developer.android.com/reference/android/arch/persistence/room/PrimaryKey.html) a unique key for each entity, you would add and annotate a primary integer key with **autoGenerate=true.**

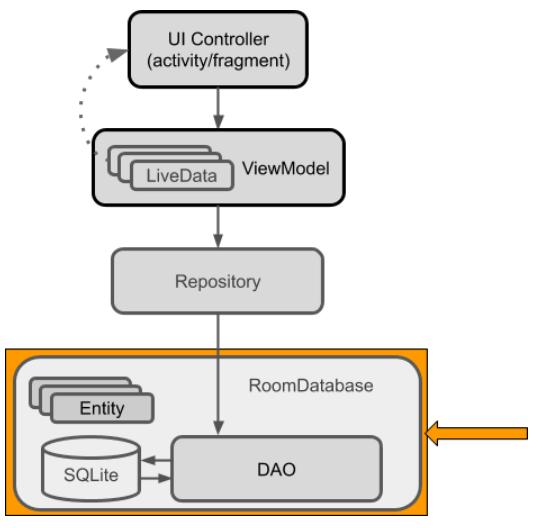
**Part 3: Create the Data Access Object (DAO)**



1. Create a new interface and call it WordDao.
2. Annotate the class declaration with @Dao to identify the class as a DAO class for Room.
3. Here is the complete WordDao interface.

@Dao  
interface WordDao {  
 @Insert(onConflict = OnConflictStrategy.IGNORE)  
 fun insert(word: Word?)  
  
 @Update  
 fun update(word: Word?)  
  
 @Delete  
 fun delete(word: Word?)  
  
 @Query("DELETE FROM word\_table")  
 fun deleteAll()  
  
 @Query("SELECT \* from word\_table ORDER BY word ASC")  
 fun getAllWords(): LiveData<List<Word>>  
}

**Part 4: Add a room database**

****

1. Create a public abstract class that extends [RoomDatabase](https://developer.android.com/reference/android/arch/persistence/room/RoomDatabase.html" \t "_blank) and call it WordRoomDatabase.

abstract class WordRoomDatabase : RoomDatabase() {}

1. Annotate the class to be a Room database.

@Database(entities = [Word::class], version = 1, exportSchema = false)  
abstract class WordRoomDatabase : RoomDatabase() {}

1. Define the DAOs that work with the database. Provide an abstract "getter" method for each @Dao.

@Database(entities = [Word::class], version = 1, exportSchema = false)  
abstract class WordRoomDatabase : RoomDatabase() {

abstract fun wordDao(): WordDao

}

1. Create the WordRoomDatabase as a [singleton](https://en.wikipedia.org/wiki/Singleton_pattern) to prevent having multiple instances of the database opened at the same time, which would be a bad thing. Here is the code to create the singleton:

companion object {  
 @Volatile  
 private var INSTANCE: WordRoomDatabase? = null

val coroutineScope = *CoroutineScope*(Dispatchers.Main)

***//Callback method here***

fun getDatabase(context: Context): WordRoomDatabase? {  
 if (INSTANCE == null) {  
 *synchronized*(WordRoomDatabase::class.*java*) **{**

***// Create database here***

**}** }  
 return INSTANCE  
 }  
}

1. Add code to create a database where indicated by the Create database here comment in the code above.

INSTANCE = databaseBuilder(  
 context.*applicationContext*,  
 WordRoomDatabase::class.*java*, "word\_database")  
 .fallbackToDestructiveMigration()

.addCallback(loadSampleData)

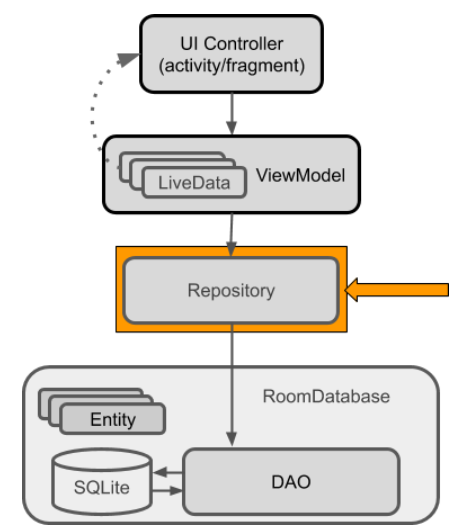
.build()

1. Implement the callback method for the **loadSampleData**.

val loadSampleData: Callback = object : Callback() {  
 override fun onOpen(db: SupportSQLiteDatabase) {  
 super.onOpen(db)  
  
 *// comment out the following block* coroutineScope.*launch*(Dispatchers.IO) **{** *// Populate the database in the background.* val dao = INSTANCE!!.wordDao()  
 dao.deleteAll()  
 var word = Word("Android")  
 dao.insert(word)  
 word = Word("IOS")  
 dao.insert(word)  
 word = Word("Symbian")  
 dao.insert(word)  
 **}** }  
}

**Part 5: Create a repository**

A Repository manages query threads and allows you to use multiple backends. In the most common example, the Repository implements the logic for deciding whether to fetch data from a network or use results cached in the local database.



1. Create a public class called WordRepository.

class WordRepository(application: Application?) {}

1. Add member variables for the DAO and the list of words.

private val mWordDao: WordDao  
private val coroutineScope = *CoroutineScope*(Dispatchers.Main)  
  
*//This method will be called in ViewModel class*var allWords:LiveData<List<Word>>? = null

1. Add an init method that gets a handle to the database and initializes the member variables.

init {  
 val db = WordRoomDatabase.getDatabase(application!!)  
 mWordDao = db!!.wordDao()  
 allWords = mWordDao.getAllWords()  
}

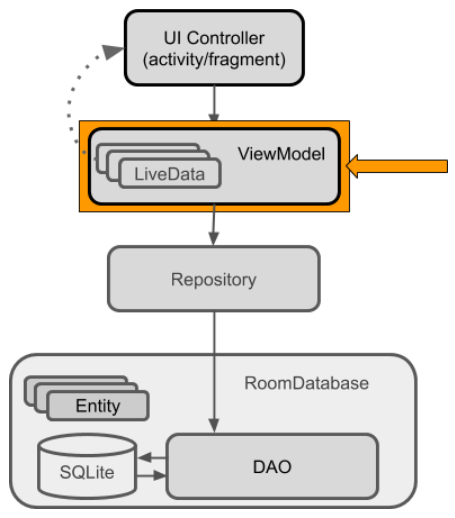
1. Add two suspend functions for inserting and deleting the data.

suspend fun asyncInsert(word:Word?) {  
 mWordDao.insert(word)  
}  
  
suspend fun asyncDelete(word:Word?) {  
 mWordDao.delete(word)  
}

1. Add two interface methods to call the suspend functions.

fun insert(word: Word?) {  
 coroutineScope.*launch*(Dispatchers.IO) **{** asyncInsert(word)  
 **}**}  
  
 fun delete(word: Word?) {  
 coroutineScope.*launch*(Dispatchers.IO) **{** asyncDelete(word)  
 **}**}

**Part 6: Create the ViewModel**



The ViewModel is a class whose role is to provide data to the UI and survive configuration changes. A ViewModel acts as a communication center between the Repository and the UI. The ViewModel is part of the [lifecycle library](https://developer.android.com/topic/libraries/architecture/lifecycle.html).

1. Create a class called WordViewModel that extends [AndroidViewModel](https://developer.android.com/reference/android/arch/lifecycle/AndroidViewModel.html" \t "_blank).

public class WordViewModel(application: Application) :

AndroidViewModel(application) {

}

1. Add two private member variables to hold a reference to the Repository and LiveData.

private val mRepository: WordRepository

1. Add a private LiveData member variable to cache the list of words.

private var allWords: LiveData<List<Word>>?

1. Add a constructor that gets a reference to the WordRepository and gets the list of all words from the WordRepository.

init {  
 mRepository = WordRepository(application)  
 allWords = mRepository.allWords  
}

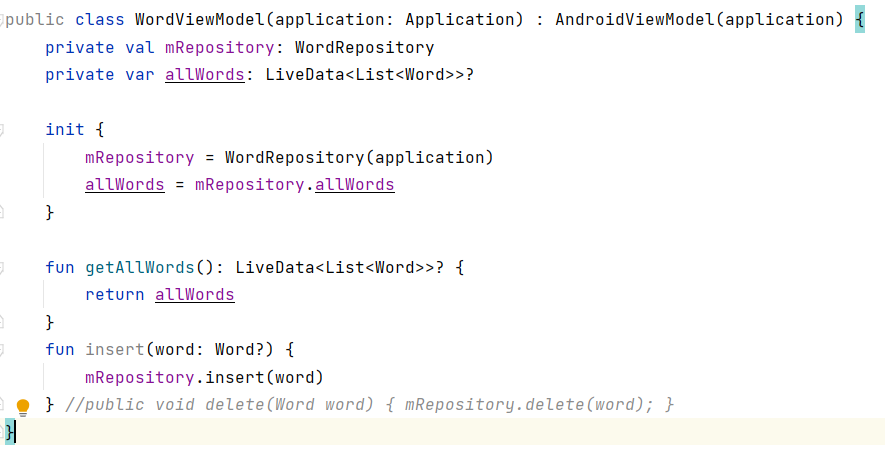
1. Add a "getter" method that gets all the words. This completely hides the implementation from the UI.

fun getAllWords(): LiveData<List<Word>>? {  
 return allWords  
}

1. Create a wrapper insert() method that calls the Repository's insert() method. In this way, the implementation of insert() is completely hidden from the UI.

fun insert(word: Word?) {  
 mRepository.insert(word)  
}

1. Here is the complete code for WordViewModel:



**Part 7: Create the Layout for RecyclerView**

1. Change the app themes (You can ignore this part):

<!-- Base application theme. -->  
    <style name="AppTheme" parent="Theme.MaterialComponents.Light.DarkActionBar">  
        <!-- Customize your theme here. -->  
        <item name="colorPrimary">@color/colorPrimary</item>  
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>  
        <item name="colorAccent">@color/colorAccent</item>  
    </style>

1. Add a style for text views in the values/themes/themes.xml file:

**<!-- The default font for RecyclerView items is too small.  
  The margin is a simple delimiter between the words. -->**

<style name="word\_title">  
 <item name="android:layout\_width">match\_parent</item>  
 <item name="android:layout\_marginBottom">8dp</item>  
 <item name="android:paddingLeft">8dp</item>  
 <item name="android:background">@android:color/holo\_orange\_light</item>  
 <item name="android:textAppearance">@android:style/TextAppearance.Large</item>  
</style>

<style name="button\_style" parent="android:style/Widget.Material.Button">  
 <item name="android:layout\_width">match\_parent</item>  
 <item name="android:layout\_height">wrap\_content</item>  
 <item name="android:background">@color/teal\_200</item>  
 <item name="android:textAppearance">@android:style/TextAppearance.Large</item>  
 <item name="android:layout\_marginTop">16dp</item>  
 <item name="android:textColor">@color/black</item>  
</style>

1. Create a layout/recyclerview\_item.xml layout as follows:

*<?*xml version="1.0" encoding="utf-8"*?>*<LinearLayout xmlns:android=  
 "http://schemas.android.com/apk/res/android"  
 android:orientation="vertical" android:layout\_width="match\_parent"  
 xmlns:tools="http://schemas.android.com/tools"  
 android:layout\_height="wrap\_content">  
  
 <TextView  
 android:id="@+id/textView"  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 style="@style/word\_title"  
 tools:text="placeholder text" />  
</LinearLayout>

1. In activity\_main.xml file, replace the TextView element with a RecyclerView element:

<androidx.recyclerview.widget.RecyclerView  
 android:id="@+id/recyclerview"  
 android:layout\_width="0dp"  
 android:layout\_height="0dp"  
 tools:listitem="@layout/recyclerview\_item"  
 android:padding="@dimen/big\_padding"  
 app:layout\_constraintBottom\_toBottomOf="parent"  
 app:layout\_constraintLeft\_toLeftOf="parent"  
 app:layout\_constraintRight\_toRightOf="parent"  
 app:layout\_constraintTop\_toTopOf="parent" />

1. Add a FAB action button at the bottom of the layout file:

<com.google.android.material.floatingactionbutton.FloatingActionButton  
 android:id="@+id/fab\_add"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:layout\_marginEnd="8dp"  
 android:layout\_marginBottom="8dp"  
 android:clickable="true"  
 app:layout\_constraintBottom\_toBottomOf="parent"  
 app:layout\_constraintEnd\_toEndOf="parent"  
 app:srcCompat="@android:drawable/ic\_input\_add"  
 tools:ignore="SpeakableTextPresentCheck" />



1. The icon in your floating action button (FAB) should correspond to the available action. In the layout/activity\_main.xml file, give the FloatingActionButton a + symbol icon.
2. Select **File > New > Vector Asset**.
3. Select **Material Icon**.
4. Click the Android robot icon in the **Icon:** field, then select the + ("add") asset.
5. In the layout/activity\_main.xml file, in the FloatingActionButton, change the srcCompat attribute to:

android:src="@drawable/ic\_add\_black\_24dp"

## Part 8: Create an Adapter and adding the RecyclerView

1. Add a class WordListAdapter that extends RecyclerView.Adapter. The adapter caches data and populates the RecyclerView with it. The inner class WordViewHolder holds and manages a view for one list item.

class WordListAdapter(val context: Context?) : RecyclerView.Adapter<WordListAdapter.WordViewHolder>() {  
 *//private var mInflater: LayoutInflater? = null* private var mWords *// Cached copy of words* : List<Word>? = null  
  
 init {  
 *//mInflater = LayoutInflater.from(context)* }  
  
 override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): WordViewHolder {  
 val mInflater = LayoutInflater.from(parent.*context*)  
  
 val itemView = mInflater.inflate(R.layout.*recyclerview\_item*, parent, false)  
 return WordViewHolder(itemView)  
 }  
  
 override fun onBindViewHolder(holder: WordViewHolder, position: Int) {  
 if (mWords != null) {  
 val current: Word = mWords!![position]  
 holder.wordItemView.setText(current.mWord)  
 } else {  
 *// Covers the case of data not being ready yet.* holder.wordItemView.*text* = "No Word"  
 }  
 }  
  
 fun setWords(words: List<Word>) {  
 mWords = words  
 notifyDataSetChanged()  
 }  
  
 *// getItemCount() is called many times, and when it is first called,  
 // mWords has not been updated (means initially, it's null, and we can't return null).* override fun getItemCount(): Int {  
 return if (mWords != null) mWords!!.size else 0  
 }  
  
 inner class WordViewHolder (itemView: View) : RecyclerView.ViewHolder(itemView) {  
 val wordItemView: TextView  
  
 init {  
 wordItemView = itemView.findViewById(R.id.*textView*)  
 }  
 }  
}

1. Declare the following attributes in the MainActivity.

class MainActivity : AppCompatActivity() {  
 private var binding: ActivityMainBinding? = null  
 private lateinit var mWordViewModel: WordViewModel  
 private var adapter:WordListAdapter = WordListAdapter(this)

}

1. Set the view binding object for the MainActivity.

override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
 *//setContentView(R.layout.activity\_main)* binding = ActivityMainBinding.inflate(*layoutInflater*)  
 setContentView(binding?.*root*)

}

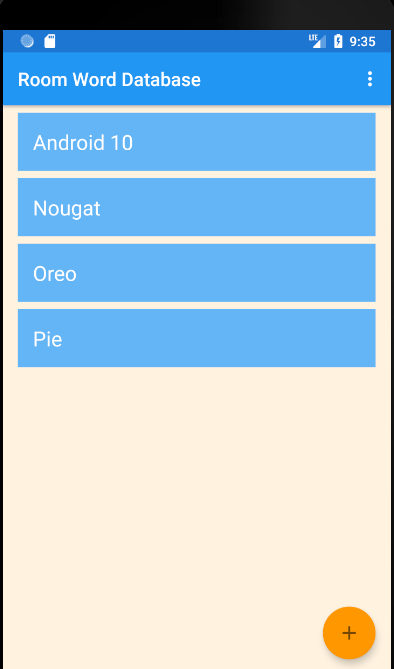
1. Implement the View Model object after the setContentView statement.

mWordViewModel = ViewModelProvider(this).get(WordViewModel::class.*java*)  
  
mWordViewModel.getAllWords()?.observe(this) **{**words:List<Word>!**->** words.*let* **{** adapter.setWords(**it:List<Word>!**)  
 **}  
}**

1. Add the RecyclerView in the onCreate() method of MainActivity:

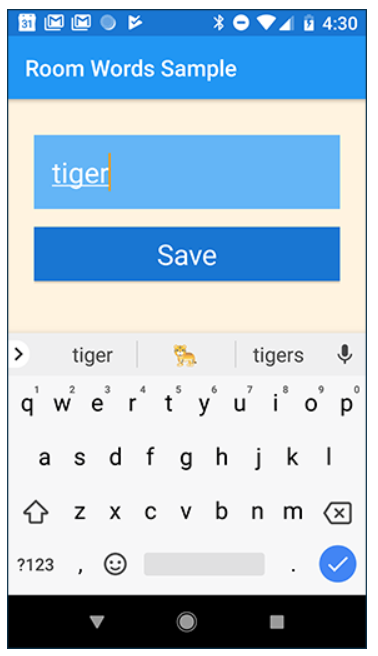
binding?.recyclerview?.setAdapter(adapter)  
binding?.recyclerview?.setLayoutManager(LinearLayoutManager(this))

1. Run the app. The initial set of words appears in the RecyclerView



## Part 11: Create an Activity for adding words

Now you will add an Activity that lets the user use the FAB to enter new words. This is what the interface for the new activity will look like:



1. Add these string resources in the values/strings.xml file:

<**string name="hint\_word"**>Word...</**string**>  
<**string name="button\_save"**>Save</**string**>  
<**string name="empty\_not\_saved"**>Word not saved because it is empty.</**string**>

1. Use the Empty Activity template to create a new activity, NewWordActivity.

1. Update the activity\_new\_word.xml file in the layout folder:

*<?***xml version="1.0" encoding="utf-8"***?>*<**LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 android:background="@color/colorScreenBackground"  
 android:orientation="vertical"  
 android:padding="24dp"**>  
  
 <**EditText  
 android:id="@+id/edit\_word"  
 style="@style/text\_view\_style"  
 android:hint="@string/hint\_word"  
 android:inputType="textAutoComplete"** />  
  
 <**Button  
 android:id="@+id/button\_save"  
 style="@style/button\_style"  
 android:text="@string/button\_save"** />  
</**LinearLayout**>

1. Implement the NewWordActivity class. The goal is that when the user presses the **Save** button, the new word is put in an Intent to be sent back to the parent Activity.

class NewWordActivity : AppCompatActivity() {  
  
 lateinit var binding:ActivityNewWordBinding  
  
 override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
  
 binding = ActivityNewWordBinding.inflate(*layoutInflater*)  
 setContentView(binding.*root*)  
  
 binding.buttonSave.setOnClickListener **{** val replyIntent = Intent()  
 if (binding.editWord.*text*.toString().*isEmpty*()) {  
 setResult(*RESULT\_CANCELED*, replyIntent)  
 } else {  
 val word = binding.editWord.*text*.toString()  
 replyIntent.putExtra(NewWordActivity.EXTRA\_REPLY, word)  
 setResult(*RESULT\_OK*, replyIntent)  
 }  
 finish()  
 **}** }  
 companion object {  
 val EXTRA\_REPLY = "com.example.android.roomwordssample.REPLY"  
 }  
}

1. In MainActivity, add the onActivityResult() callback for the NewWordActivity. If the activity returns with RESULT\_OK, insert the returned word into the database by calling the insert() method of the WordViewModel.

**lateinit var startForResult: ActivityResultLauncher<Intent>**

override fun onCreate(savedInstanceState: Bundle?) {

binding = ActivityMainBinding.inflate(*layoutInflater*)  
 setContentView(binding?.*root*)  
  
**startForResult = registerForActivityResult<Intent, ActivityResult>(  
 ActivityResultContracts.StartActivityForResult()) {result:ActivityResult ->  
  
 if (result.*resultCode* == *RESULT\_OK*) {  
 val data = result.*data* val newWord = data!!.getStringExtra(NewWordActivity.EXTRA\_REPLY)  
 Log.d("newword", newWord!!)  
 val word = Word(newWord)  
 mWordViewModel.insert(word)  
 } else {  
 Toast.makeText(  
 *applicationContext*,  
 R.string.*empty\_not\_saved*,  
 Toast.*LENGTH\_LONG* ).show()  
 }  
 }**

}

1. In MainActivity,start NewWordActivity when the user taps the FAB. Replace the code in the FAB's onClick() click handler with the following code:

binding?.fabAdd?.setOnClickListener **{**val intent = Intent(this@MainActivity, NewWordActivity::class.*java*)  
  
 startForResult.launch(intent)  
**}**

1. Run your app. When you add a word to the database in NewWordActivity, the UI automatically updates.

1. Add a word that already exists in the list. What happens? Does your app crash? Your app uses the word itself as the primary key, and each primary key **must** be unique.
2. In the WordDao interface, change the annotation for the insert() method to:

@Insert(onConflict = OnConflictStrategy.***IGNORE***)  
**void** insert(Word word);

1. Run your app again and try adding a word that already exists. What happens now?