Lesson 4 (Part 3)

Objectives

- Explain the components needed to set up a RecyclerView widget
- Display data in a local SQLite database in a RecyclerView widget by writing a RecyclerView adapter class
- Write code to Delete a RecyclerView ViewHolder by means of swiping

The Android/Java you need to know

Gradle and Configuration Files

Gradle is the software component that manages the build process for an android app.

The build process begins from the source code and ends with the APK file.

The APK file can then be installed on any Android phone.

You may obtain the APK file in Android studio using

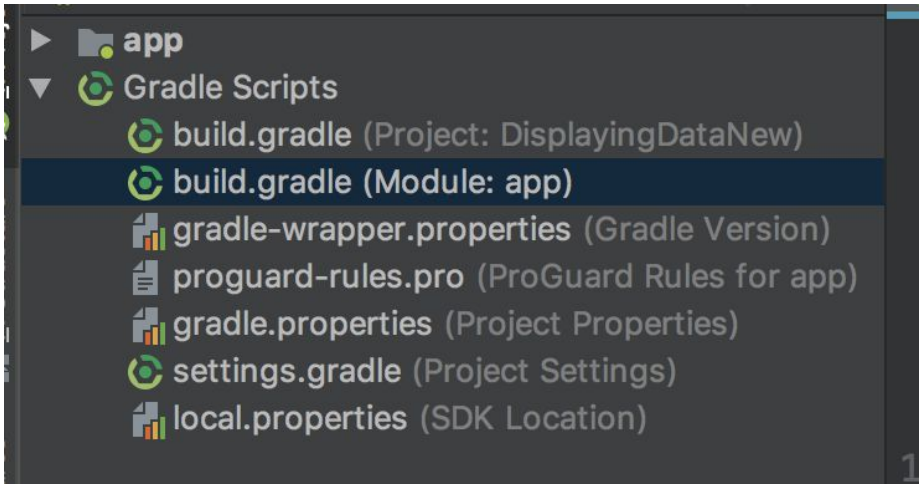
Build → Build APK(s)

More information on the build process here:

https://developer.android.com/studio/build/

Settings for the build process are stored in two build.gradle files:

- the project-level file
- the module-level file



Gradle module-level settings

Often the project-level default settings are sufficient.

Hence, we usually have to modify the module-level settings only.

The first part (Lines 1 - 20) shows information such as

- Minimum API level
- Compile API level
- Target API level

You may adjust these settings if you are aiming for certain API levels.

```
1   apply plugin: 'com.android.application'
2
3   android {
4       compileSdkVersion 28
5       defaultConfig {
6           applicationId "com.example.norman_lee.myapplication"
7           minSdkVersion 15
8           targetSdkVersion 28
9           versionCode 1
10          versionName "1.0"
11          testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
12      }
13      buildTypes {
14          release {
15              minifyEnabled false
16              proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-ru
17          }
18      }
19  }
20
```

The following part (Line 21 onwards) shows the dependencies that your app has.

```
21  dependencies {
22      implementation fileTree(dir: 'libs', include: ['*.jar'])
23      implementation 'com.android.support:appcompat-v7:28.0.0-rc02'
24      implementation 'com.android.support.constraint:constraint-layout:1.1.3'
25      implementation 'com.android.support:design:28.0.0-rc02'
26      implementation 'com.android.support:recyclerview-v7:28.*'
27      implementation 'com.android.support:cardview-v7:28.0.0'
28      testImplementation 'junit:junit:4.12'
29      androidTestImplementation 'com.android.support.test:runner:1.0.2'
30      androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.2'
31  }
```

Lines 26 and 27 are not part of the default dependencies generated from a fresh project. They were added in, and adding them in downloads the packages if you are doing it for a first time.

Strategy Design Pattern

In the strategy design pattern, parts of the behaviours of an object is handed over to other objects. This is known as **delegation**.

This provides flexibility at run-time as you can change those behaviours.

```
public abstract class Duck {

    private FlyBehavior flyBehavior;
    private QuackBehavior quackBehavior;
    String name;

    public Duck(){
    }

    public Duck(String name){
        this.name = name;
    }

    public void setFlyBehavior(FlyBehavior flyBehavior) {
        this.flyBehavior = flyBehavior;
    }

    public void setQuackBehavior(QuackBehavior quackBehavior) {
        this.quackBehavior = quackBehavior;
    }

    public void performFly(){
        flyBehavior.fly();
    }

    public void performQuack(){
        quackBehavior.quack();
    }

    public abstract void display();
}
```

In the abstract class above, the delegation happens as follows

- The flying behaviour is delegated to a **`FlyBehavior`** object
- The quacking behaviour is delegated to a **`QuackBehavior`** object

For the FlyBehavior, we implement different objects that represent different behaviour.

```
interface FlyBehavior {
   void fly();
}
```

```
class FlapWings implements FlyBehavior {
   @Override
   public void fly() {
       System.out.println("Flapping my Wings");
   }
}
```

Implement a class **`CannotFly`** that implements **`FlyBehavior`**.

The **`fly()`** method prints outs "**`I cannot fly :(`**"

Similarly, for **`QuackBehavior`** objects:

```
public interface QuackBehavior {
   void quack();
}
```

```
public class LoudQuack implements QuackBehavior {
   @Override
   public void quack() {
       System.out.println("QUACK");
   }
}
```

Finally, we subclass Duck with our own object.

```
public class MallardDuck extends Duck {

   MallardDuck(String name){
       super(name);
   }

   @Override
   public void display() {
       System.out.println("I am " + name + ", the Mallard Duck");
   }
}
```

And we can run our **`MallardDuck`** object and set their behaviours at run-time:

```
public class TestDuck {

   public static void main(String[] args){

       Duck duck = new MallardDuck("Donald");
       duck.setFlyBehavior(new FlapWings());
       duck.setQuackBehavior(new LoudQuack());
       duck.display();
       duck.performFly();
       duck.performQuack();
   }
}
```

We see here the **flexibility of composition over inheritance**.

We develop our duck behaviours independent of the type of duck.

The behaviour of the duck is delegated to separate objects.

You assemble your specific duck at run-time,

which gives you the flexibility to change its behaviour if needed.

Adapter Design Pattern

The word **interface** is an overloaded word

- in Java terminology it would mean a type of class with method signatures only
- it could also mean the set of methods that a class allows you to access
  (think of ‘user interface’)

An adapter design pattern converts the interface of one class into another that a client class expects.

Adapter Design Pattern Example

You have an interface **`Duck`** and a class **`MallardDuck`** that implements this interface.

```
public interface Duck {
   void quack();
   void fly();
}
```

```
public class MallardDuck implements Duck {
   @Override
   public void quack() {
       System.out.println("Mallard Duck says Quack");
   }

   @Override
   public void fly() {
       System.out.println("Mallard Duck is flying");
   }
}
```

Then you have a client that loops through all ducks and make them fly and quack.

```
import java.util.ArrayList;
public class DuckClient {

   static ArrayList<Duck> myDucks;

   public static void main(String[] args){
       myDucks = new ArrayList<>();
       myDucks.add( new MallardDuck());
       makeDucksFlyQuack();
   }

   static void makeDucksFlyQuack(){
      for(Duck duck: myDucks){
               duck.fly();
               duck.quack();
      }
   }
}
```

Now you have a **Turkey** interface.

```
public interface Turkey {

   public void gobble();
   public void fly();
}
```

How might we allow **Turkey** objects to be used by the same client?

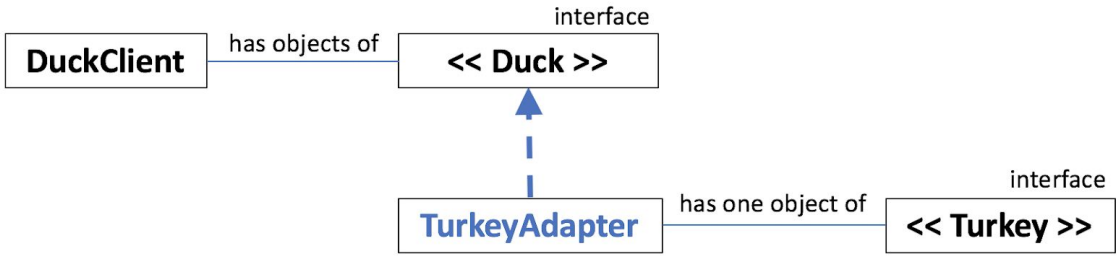
We write an adapter class that

- has the same Duck interface and
- takes in a Turkey object

```
public class TurkeyAdapter implements Duck {

   Turkey turkey;

   TurkeyAdapter(Turkey turkey){
       this.turkey = turkey;

   }
   @Override
   public void quack() {
       //implement this
   }

   @Override
   public void fly() {
       //implement this
   }
}
```

This material was taken from “HeadFirst-Design Patterns”

Explaining the Duck/Turkey Adapter Example



What is a RecyclerView?

Suppose you have a collection of similar data e.g.

- Images with descriptions
- Chat messages with sender's name

… and you want to display them in your app.

The **RecyclerView** widget allows the user to scroll through the data.
This is done by loading each data item onto its own item in RecyclerView.
A typical RecyclerView display is shown below.

CardView

A useful widget that can display data in RecyclerView is the **CardView** widget.

To use CardView, ensure that you have the following dependency in your module-level gradle file:

```
implementation 'com.android.support:cardview-v7:28.0.0'
```

CardView gives the “card look” to each item.

- You can change attributes to tweak the look of the cards.
- You then specify the layout of widgets within a CardView.

The following is an overview of an XML file specifying a CardView and the layout within. Details have been removed from most widgets.

```
<android.support.v7.widget.CardView
   android:id="@+id/cardViewItem"
   app:cardPreventCornerOverlap="false"
   cardCornerRadius="5dp"
   cardMaxElevation="1dp"
   cardElevation="1dp"
   cardUseCompatPadding="true"
   android:layout_width="match_parent"
   android:layout_height="100dp"
   android:layout_margin="16dp">

   <RelativeLayout
       android:id="@+id/ard"
       android:layout_width="match_parent"
       android:layout_height="match_parent">

       <ImageView />
       <TextView />
       <TextView />
       <TextView />
   </RelativeLayout>

</android.support.v7.widget.CardView>
```

How to implement RecyclerView

To use RecyclerView in your app,

Step 1. ensure that you have the following dependency in your module-level gradle file

```
implementation 'com.android.support:recyclerview-v7:28.*'
```

Step 2. Include the following widget tag in the Activity layout where you want to have the recyclerView.

```
<android.support.v7.widget.RecyclerView
   android:id="@+id/charaRecyclerView"
   android:layout_width="match_parent"
   android:layout_height="match_parent" />
```

Step 3. Assuming each data item is stored in a CardView, design the layout of each data item.

Step 4. Decide the source of your data:

- Stored in the res folder
- SQLiteDatabase
- Cloud Database
- etc

This lesson shows you how to use data from a local SQLiteDatabase.

You would have written a Database Helper class.

Step 5. Write an Adapter class that extends the **`RecyclerView.Adapter<VH>`** class.

This class takes in your data source and is called by the Android runtime to display the data on the RecyclerView widget. This class also references the data item that you designed in step 3.

This will be explained in the next section.

Step 6 continues next page …

Step 6. In the java file for your activity, write code for the following

- Get a reference to the recyclerView widget using findViewById()
- Get an instance of an object that points to your dataSource
- Instantiate your Adapter
- Attach the adapter to your recyclerView widget
- Attach a Layout manager to your recyclerView widget. A LayoutManager governs how your widgets are going to be displayed. Since we are scrolling up and down, we will just need a LinearLayoutManager.

The sample code is here. You will need to adapt the code a little.

```
recyclerView = findViewById(R.id.charaRecyclerView);
dataSource = ??? ;
charaAdapter = new CharaAdapter(this, dataSource );
recyclerView.setAdapter(charaAdapter);
recyclerView.setLayoutManager(new LinearLayoutManager(this));
```

This way of coding shows you how **delegation** is performed.

Delegation is the transferring of tasks from one object to a related object.

The **`RecyclerView`** object delegates

- the role of retrieving data to the **`RecyclerView.Adapter`** object.
- the role of managing the layout to the **`LinearLayoutManager`** object

Thus the RecyclerView object makes use of the Strategy Design Pattern.

Writing The RecyclerView Adapter - Static Inner Class

The Recycler View adapter class is the adapter class between the RecyclerView widget and the object containing your source of data.

Your RecyclerView Adapter should extend the **`RecyclerView.Adapter<VH>`** class.
`VH` is a generic class that subclasses **`RecyclerView.ViewHolder`**.
This is an **abstract class** without abstract methods.
Hence, Android is forcing you to subclass this class to use its methods.

This class is meant to hold references to the widgets in each data item layout.
Typically, we will write such a class as an inner class within the recyclerView adapter.

Hence, the classes are declared in the following way.

```
public class CharaAdapter extends
RecyclerView.Adapter<CharaAdapter.CharaViewHolder>{

   //code not shown
   static class CharaViewHolder extends RecyclerView.ViewHolder{
        //code not shown
   }
}
```

Having designed your CardView layout for each data item, **`CharaViewHolder`** will contain instance variables that are meant to hold references to the widgets on the layout.
The references are obtained by calling **`findViewById()`** within the constructor.

Writing the RecyclerView Adapter - write the constructor and override three methods

The **constructor** should take in

- a Context object
- Object for your data source

The context object is used to get a layout inflator object to be used in **`onCreateViewHolder()`**.

`RecyclerView.Adapter<VH>` is an abstract class and you have to override three abstract methods.

`onCreateViewholder()` is called by the run-time each time a new data item is added.
In the code recipe below:

- The CardView layout is inflated
- A reference to the layout in memory is returned **`itemView`**
- This reference **`itemView`** is passed to the constructor of **`CharaViewHolder`**
- **`CharaViewHolder`** uses this reference to get references to the individual widgets in the layout

Here's a typical recipe:

```
public CharaViewHolder onCreateViewHolder(@NonNull ViewGroup
viewGroup, int i) {
   View itemView = mInflater.inflate(R.layout.layout, viewGroup,
false);
   return new CharaViewHolder(itemView);
}
```

`onBindViewHolder()` is meant to

- get the appropriate data from your data source
- attach it to the widgets on each data item, according to the adapter position.

Hence, the data on row 0 of a table goes on position 0 on the adapter and so on.

`getItemCount()` is meant to return the total number of data items. Hence, if you return 0, nothing can be seen on the RecyclerView.

Seeing the connection

The Recycler View adapter class is the adapter class between the RecyclerView widget and the object containing your source of data.

Compare the RecyclerView implementation with the Duck/Turkey example

Example	**RecyclerView component**
Duck and DuckClient	
TurkeyAdapter	
Turkey	

Getting Each Item to Respond to Clicks

This is not in the list of TODOs, but it would be instructive to think about how it can be done.

Since we extend **`RecyclerView.ViewHolder`**, we have access to the parent class' methods. One method is **`getAdapterPosition()`**, which displays the ViewHolder's position on the Recycler View.
Use this method to display a toast when each ViewHolder is clicked.

Option 1. Since a reference to the CardView layout is passed to the ViewHolder class, then you may call **`setOnClickListener`** on this reference within the constructor, and pass to it an anonymous class in the usual way.

Option 2. **`CharaViewHolder`** class can implement the **`View.OnClickListener`** interface. Then **`onClick`** has to be implemented as an instance method.
You still need to call **`setOnClickListener`** on the reference to the CardView layout.
What object do you pass to **`setOnClickListener`**?

Swiping To Delete

We are able to write code to delete a particular ViewHolder when it is swiped left/right.
The code recipe is to create an instance of **`ItemTouchHelper`**
and attach the RecyclerView instance to it.

```
ItemTouchHelper itemTouchHelper
              = new ItemTouchHelper(simpleCallback);
itemTouchHelper.attach(recyclerView);
```

The constructor takes in an object that extends the **`ItemTouchHelper.SimpleCallback`** abstract class.
To use this class,

- Pass the direction of swiping that you want to detect to its constructor
- Override **`onSwipe()`**

From the documentation, the directions are specified via constants.
As you are going to use this object only once,
an acceptable practice is to use an anonymous abstract class.
As we are coding for swiping, we do not write any other code in **`onMove()`**.

```
ItemTouchHelper.SimpleCallback simpleCallback = new
ItemTouchHelper.SimpleCallback(0, ItemTouchHelper.LEFT |
ItemTouchHelper.RIGHT ) {
   @Override
   public boolean onMove(@NonNull RecyclerView recyclerView,
@NonNull RecyclerView.ViewHolder viewHolder, @NonNull
RecyclerView.ViewHolder viewHolder1) {
       return false;
   }

   @Override
   public void onSwiped(@NonNull RecyclerView.ViewHolder
viewHolder, int i) {

   }
}
```

Two parameters are passed to **`onSwiped()`**:

- an instance of the ViewHolder that is currently being swiped
- the direction (change the variable name of the autogenerated code …)

The tasks are

- **Downcast** the ViewHolder object so that you can use the instance variables or methods that you have defined
- Call your database helper with the required information to delete the particular row in the database
- Display any other UI message e.g. a toast message saying a deletion has been happening
- Notify the RecyclerView adapter that the database has an item removed
  (Where did the **`getAdapterPosition()`** method come from?)

```
CharaAdapter.CharaViewHolder charaViewHolder =
(CharaAdapter.CharaViewHolder) viewHolder;

String name = charaViewHolder.textViewName.getText().toString();
charaDbHelper.deleteOneRow(name);

Toast.makeText(RecylerViewActivity.this, "Deleting " +
       name, Toast.LENGTH_LONG).show();

charaAdapter.notifyItemRemoved(viewHolder.getAdapterPosition());
```

Building Your App

TODO 9.1 - In MainActivity, add an Options Menu Item to bring your users to the RecyclerView activity.

When this is done, you should see an empty screen. You may add an additional text view widget to the RecyclerView activity layout to gain the confidence that you have done this correctly.

TODO 9.2 - 9.6 - Complete CharaAdapter

To do this, you should

- Complete the inner class CharaViewHolder
- Complete the constructor
- Complete onCreateViewHolder
- Complete onBindViewHolder
- Complete getItemCount to return the size of the data

You may add logcat statements to see when onCreateViewHolder and onBindViewHolder are invoked

TODO 9.7 Complete RecyclerViewActivity

Add the code so that your RecyclerView widget has an adapter and a linearLayout manager. At this stage, if you have added logcat statements above, add sufficient data so that your screen is filled. You may scroll up and down and observe the logcat statements to see how the adapter callbacks are invoked.

TODO 9.8 Complete the code to delete by swiping left and right

Explained above.