Lesson 4.1 - Pokédex

This is an updated version of Lesson 4 since 2019.

Objectives

- Calling intents using startActivityForResult() in an android app
- Implement a RecyclerView in an Android app
- Describe the use of static nested classes
- Describe Delegation and the Strategy Design Pattern
- Describe the Adapter Design Pattern

The Android/Java that you need to know

Recall Lesson 2 & 3

We might get bogged down with specific details in the code, but we should also gain an overview of the Android framework and some advanced features in Java.

For each feature described, state the part of the android framework that you can implement.

Feature	**Framework component**
Bring the user from one Activity to another Activity	Explicit intent
To validate that your app behaves as it should when a user interacts with it	Instrumented Testing
Bring the user from the app to another app with a specific function	Implicit intent
The series of callbacks as an Activity is created and/or destroyed	onCreate, onDestory, onStart, onRestart, onStop, onPause
To run long tasks in another thread	ExecutorService
Store data to make it available at another point in time	SharedPreferences
To validate that code components (e.g. methods) in your app behaves as it should	Unit Testing
To develop reusable code by subtyping polymorphism	Delegation
To develop reusable code by parametric polymorphism	Generics
To develop reusable code by subtyping polymorphism and parametric polymorphism	Generic Inteface (page 124)

Static Nested Classes

Recall that a class definition can contain **nested classes**.

```
public class OuterClass {
   // code not shown

   class InnerClass{
       //code not shown
   }

}
```

This is typically done when you have classes that logically depend on the outer class and are used together with the outer class.

By declaring a nested class as static, it is known as a **static nested class.**

- It can only access static variables and methods in the outer class.
- It can be instantiated without an instance of the outer class.

A static nested class behaves like a top-level class and is a way to organize classes that are used only by some other classes.
One reason for having a static nested class is to have a model class to store data.

Recall the Singleton design pattern

Recall that the singleton design pattern allows only one instance of a class to exist.

This is done by

- Making the constructor private
- The sole instance is stored in a private static variable
- Using a static factory method to return an instance

```
public class Singleton{

    private static Singleton singleton;

    private Singleton(){
         //any tasks you need to do here
    }

    public static Singleton getInstance(){

        if(singleton == null){
            singleton = new Singleton();
        }

        return singleton;
     }

     //other methods in your class

}
```

Static inner classes can be used to model data

The following example is from the app that you will build in the lesson.

`DataSource` is a class that is meant to contain data that will be displayed in the RecyclerView.

A private ArrayList variable is declared to hold instances of **`CardData`**, a static inner class.

Each instance of **`CardData`** is meant to hold information for one image.

Bear in mind that such static classes cannot access non-static variables of the enclosing class.

```
public class DataSource {

   private ArrayList<CardData> dataArrayList;

   //rest of class not shown --

   static class CardData{

       private String name;
       private String path;
       //constructors and getters not shown

   }

}
```

Intents - startActivityForResult()

Recall that

- **Explicit intents** bring you from one activity to another. Data can be passed during this process.
- **Implicit intents** bring you from one activity to a component in your app. You specify the kind of component you want, and the android runtime fetches what is available.

In both cases, you launched the “destination” by invoking startActivity().

By invoking **`startActivityForResult()`**,
you expect the destination component to return a result.

Explicit Intent - startActivityForResult()

Step 1. Declare your request code, a final static integer variable that contains a unique integer that identifies your particular intent.

This is necessary as your activity could have more than one call to **`startActivityForResult()`**

```
final int REQUEST_CODE_IMAGE = 1000;
```

Step 2. Declare an explicit intent in the usual way.

Then invoke **`startActivityForResult()`** with two arguments

- The intent
- The request code

```
Intent intent = new Intent(MainActivity.this, DataEntry.class);
startActivityForResult(intent, REQUEST_CODE_IMAGE);
```

Step 3. In the destination activity, the user should interact with it.

Step 4. A user action (e.g. clicking a button) brings the user back to the origin activity.

The following code initiates this process.

```
Intent returnIntent = new Intent();
returnIntent.putExtra(KEY, value); //optional
setResult(Activity.RESULT_OK, returnIntent);
finish();
```

The first argument of **`setResult()`** is either

- **`Activity.RESULT_OK`** if the user has successfully completed the tasks
- **`Activity.RESULT_CANCELED`** if the user has somehow backed out

Hence, this code may be written twice, one for each scenario described above.

If you have data to transfer, you are reminded that you can use the **`putExtra()`** method above. (Recall Lesson 2).

Step 5. Back in the origin activity, override the callback `onActivityResult()` to listen out for the result and carry out the next task. Note the sequence of if-statements.

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent
data) {

   if (requestCode == REQUEST_CODE_IMAGE) {
       if(resultCode == Activity.RESULT_OK){

           //if you use putExtra in Step 4, then you need this step
           double value = data.getDoubleExtra(DataEntry.KEY,
defaultValue);
           Toast.makeText(this,"Message",Toast.LENGTH_LONG).show();
       }
       if (resultCode == Activity.RESULT_CANCELED) {
           //Write your code if there's no result
       }
   }
}
```

If you have data transferred from step 4, you may retrieve this data on the intent object passed to this callback using the **`getDoubleExtra()`** method or other suitable methods(Recall Lesson 2).

Further reading

- https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/unit-1-get-started/lesson-2-activities-and-intents/2-1-c-activities-and-intents/2-1-c-activities-and-intents.html#gettingdatabackfromactivity
- https://developer.android.com/training/basics/intents/result

Implicit Intents - opening the image gallery

Attention if you are using the android emulator. To have some images for you to select, one easy way is to use the emulator to take some pictures first. I leave you to discover the best solution for yourself.

The implicit intents to write can be found at the following **Common Intents** reference
https://developer.android.com/guide/components/intents-common#java
To get the example code to retrieve an image, go to the **File Storage → Retrieve a Specific Type of File** section.

In case the website is not accessible, here is the sample code,

Example intent to get a photo:

KOTLIN JAVA

```java
static final int REQUEST_IMAGE_GET = 1;

public void selectImage() {
    Intent intent = new Intent(Intent.ACTION_GET_CONTENT);
    intent.setType("image/*");
    if (intent.resolveActivity(getPackageManager()) != null) {
        startActivityForResult(intent, REQUEST_IMAGE_GET);
    }
}

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode == REQUEST_IMAGE_GET && resultCode == RESULT_OK) {
        Bitmap thumbnail = data.getParcelable("data");
        Uri fullPhotoUri = data.getData();
        // Do work with photo saved at fullPhotoUri
        ...
    }
}
```

An Alternative to `startActivityForResult()`(Will not be covered in 2022 Spring)

In the caller activity

```
ActivityResultLauncher<Intent> launcher2 =
        registerForActivityResult(new StartActivityForResult(),
                new ActivityResultCallback<ActivityResult>() {
                    @Override
                    public void onActivityResult(ActivityResult result) {
                        if (result.getResultCode() != Activity.RESULT_OK) {
                            // continue with the result
                        } else {
                            // report the failure
                        }
                    }
                });
```

In the `onCreate()` of the caller activity

```
Intent intent = new Intent(MainActivity.this, Activity2.class);
launcher2.launch(intent);
```

In the sub activity, follow step 4 of the `startActivityForResult()` approach.

For implicit intents and other contract templates, refer to

https://developer.android.com/training/basics/intents/result

https://developer.android.com/reference/androidx/activity/result/contract/ActivityResultContracts

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

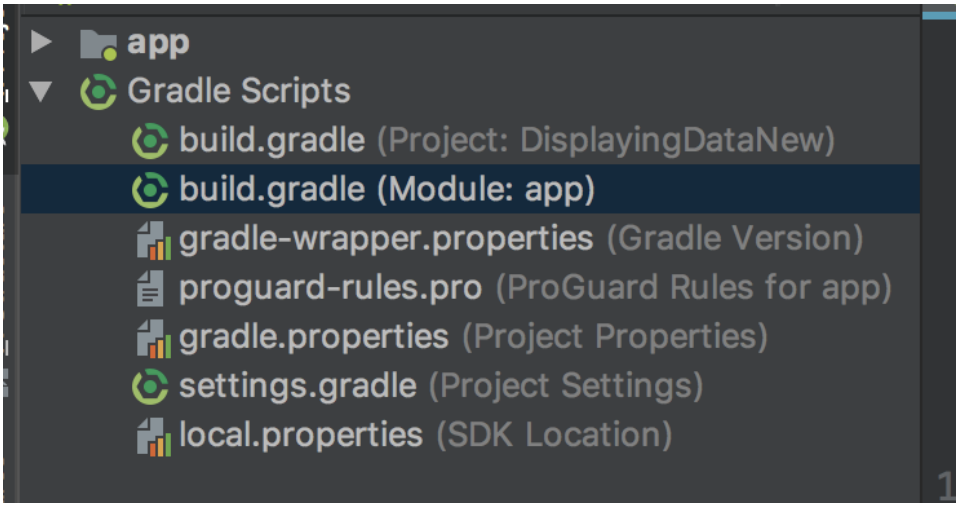
Build → Build APK(s)

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. At the point of writing this note, Google recommends to use 30+ as compiledSdkVersoin and targetSdkVersion to meet the publishing requirement in Google Play.

```
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 the first time. If you are upgrading the `compileSdkVersion`, your IDE might recommend you to upgrade the dependencies accordingly. Please use the refactor option.

Strategy Design Pattern

In the strategy design pattern, parts of the behaviours of an object are 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 out "**`I cannot fly :(`**"

```
class CannotFly implements FlyBehavior {
  @Override
  public void fly() {
    System.out.println("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();  I am Donald the Mallard Duck
       duck.performFly();  Flapping my Wings
       duck.performQuack();  QUACK
   }
}
```

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 makes 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?

Adapter design pattern = make some object compatible with another

We write an adapter class that

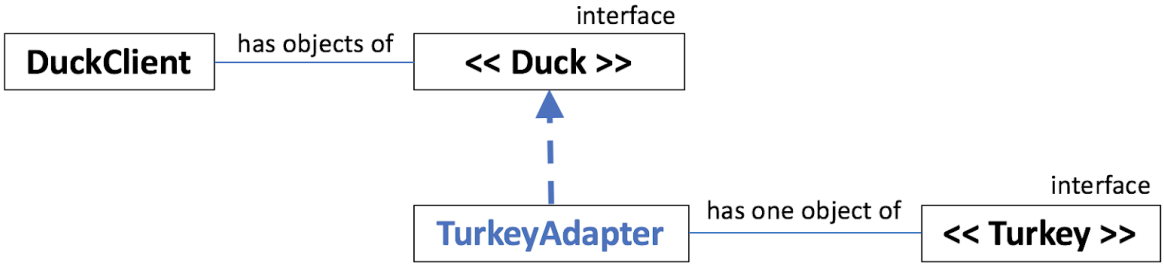
- has the same Duck interface and
- takes in a Turkey object

TurkeyAdapter is a Duck now

```
public class TurkeyAdapter implements Duck {

   Turkey turkey;

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

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

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

This material was taken from "HeadFirst-Design Patterns"

Explaining the Duck/Turkey Adapter Example




```
TurkeyAdapter turkeyAdapter = new TurkeyAdapter(new Turkey () {
  @Override
  public void gobble() {
  }

  @Override
  public void fly() {
  }
}

* This is like an anonymous class from Android Lesson 1
```

```
public class ClientMain {

    public static void main(String[] args) {
        Duck duck = new MallardDuck();
        Turkey turkey = new WildTurkey();

        Duck turkeyAdapter = new TurkeyAdapter(turkey);
        System.out.println("----Turkey");
        turkey.gobble();
        turkey.fly();

        System.out.println("----Duck");
        testIt(duck);

        System.out.println("----Turkey Adapter");
        testIt(turkeyAdapter);
    }

    public static void testIt(Duck duck) {
        duck.quack();
        duck.fly();
    }
}
```

*WildTurkey here is presumably a class like MallardDuck

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.google.android.material:material:1.0.0'
```

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.

```
<androidx.cardview.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 />
   </RelativeLayout>

</androidx.cardview.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 'androidx.recyclerview:recyclerview:1.0.0'
```

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

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

Step 3. In AndroidManifest.xml, add the attribute **android:exported=”true”** to the Activity element in which the RecyclerView is used. (Compulsory for SDK 31+)

Step 4. Assuming each data item is stored in a CardView, design the layout of each data item. CardView is an XML file!

Step 5. 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 6. 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 7 continues on the next page …

Step 7. 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.

A **`GridLayoutManager`** is also available.

Writing The RecyclerView Adapter - Static Inner Class

The RecyclerView 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
   }    No override here
}       Has a constructor
```

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 RecyclerView 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	RecyclerView
TurkeyAdapter	RecyclerView.Adapter<VH>
Turkey	ViewHolder

ViewHolder = charaViewHolder = item that contains each individual cardView

RecyclerView.Adapter<VH> = charaAdapter = adapt the information passed for flexibility

RecyclerView = recyclerView = client that loops through to set a certain stuff
(Uses strategy design pattern to manage the things)

Ways of Storing Data

You have learnt in Lesson 2 how to use **`SharedPreferences`** to store data.

There are many other options available to store data, some of which include:

- The app's internal file storage
- Storage space external to the app (e.g. the SD card)
- An SQLite database linked to your app
- Firebase realtime database

`SharedPreferences` is mainly for storing small amounts of data.
For larger amounts of data, e.g. images, you would have to consider other options, such as SQLite, the file storage options or Firebase database.

You can read more about your storage options at Chapter 9.0 of the Android developer fundamentals version 2.

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 RecyclerView.
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

This is also not in the list of TODOs.

However, it would be instructive to consider how we might delete entries from the RecyclerView.

We are able to write code to delete a particular ViewHolder when it is swiped left/right.

First, add a method in **`DataSource`** to delete any particular entry, if one is not present.

```
void removeData(int i){
    dataArrayList.remove(i);
}
```

The code recipe is to create an instance of **`ItemTouchHelper`** and attach the RecyclerView instance to it.

```
ItemTouchHelper itemTouchHelper
              = new ItemTouchHelper(simpleCallback);
itemTouchHelper.attachToRecyclerView(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) {
            //code to delete the view goes here
    }
}
```

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 …)

Within **`onSwiped()`**, the tasks are

- **Downcast** the ViewHolder object so that you can use the instance variables or methods that you have defined
- Call your dataSource 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;
int position = charaViewHolder.getAdapterPosition();
dataSource.removeData(position);
//Write code to display a Toast if you'd like
charaAdapter.notifyDataSetChanged();
```

For your reference, the entire sequence of code is shown below.

```
charaAdapter = new CharaAdapter(this, dataSource);
recyclerView.setAdapter(charaAdapter);
recyclerView.setLayoutManager( new GridLayoutManager(this,3) );

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) {

        CharaAdapter.CharaViewHolder charaViewHolder = (CharaAdapter.CharaViewHolder) viewHolder;
        int position = charaViewHolder.getAdapterPosition();
        dataSource.removeData(position);
        charaAdapter.notifyDataSetChanged();

    }
};

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

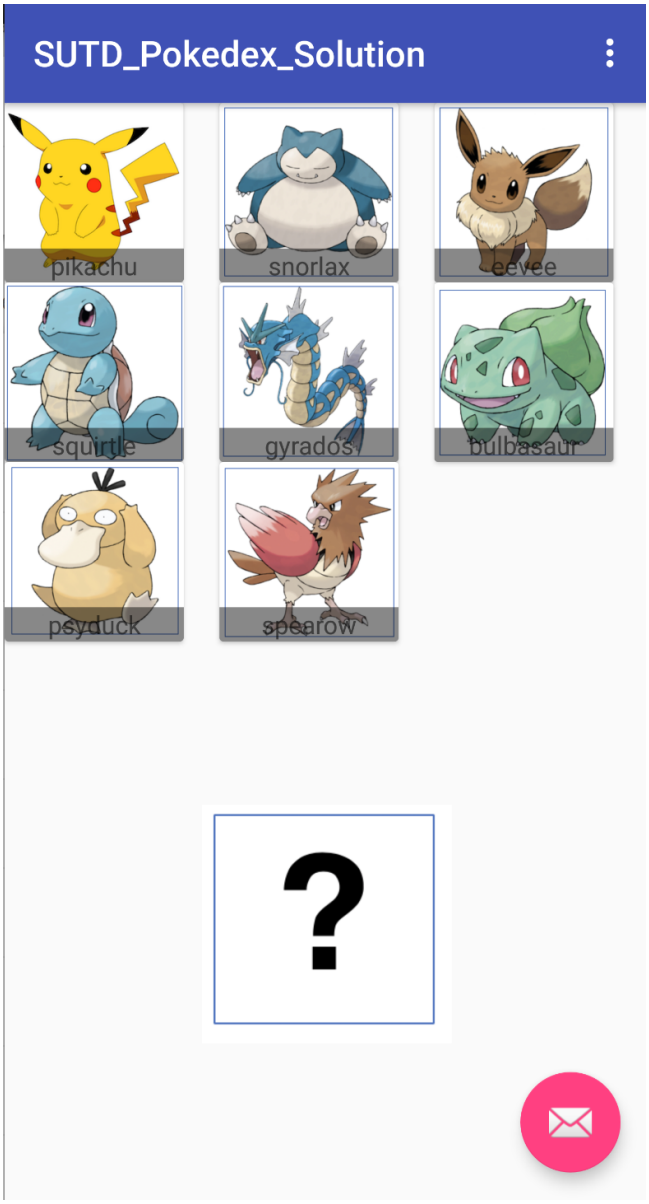
Building Your App

We will first build a RecyclerView using images placed in the drawables folder (TODOs 11.x). With this, we are not able to add images to the RecyclerView.

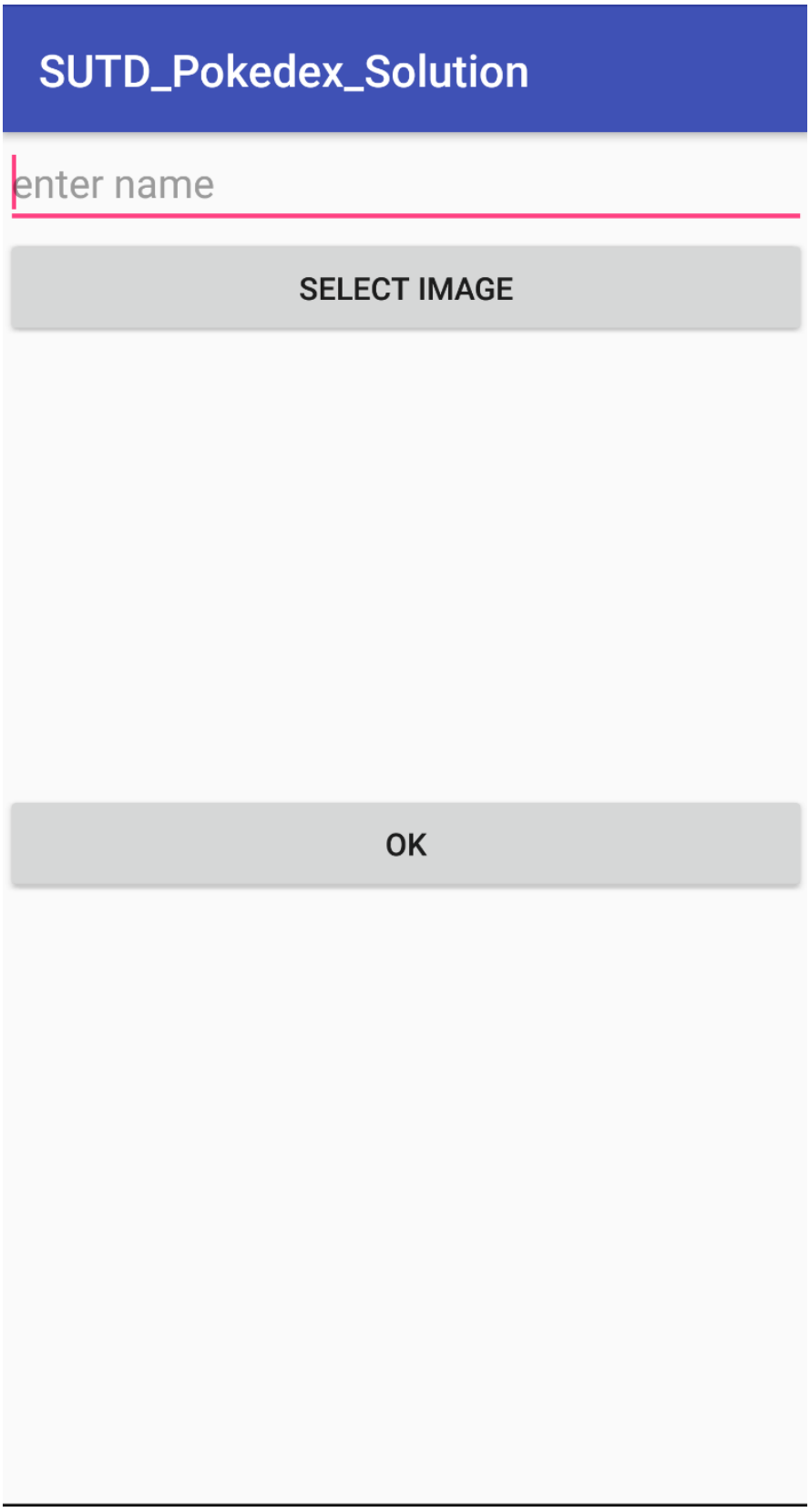
We will then be able to add images to the RecyclerView by using images selected from the camera gallery. The information is stored in the app's internal file storage. (TODOs 12.x).

Screenshots of the app

MainActivity shows the RecyclerView, an ImageView and a FloatingActionButton.
The ImageView is used to display the newest image added to the RecyclerView.



When the FloatingActionButton is clicked, it brings the user to DataEntryActivity, where the user can enter the name of the image, and select an image from the Image gallery by clicking **SELECT IMAGE**.



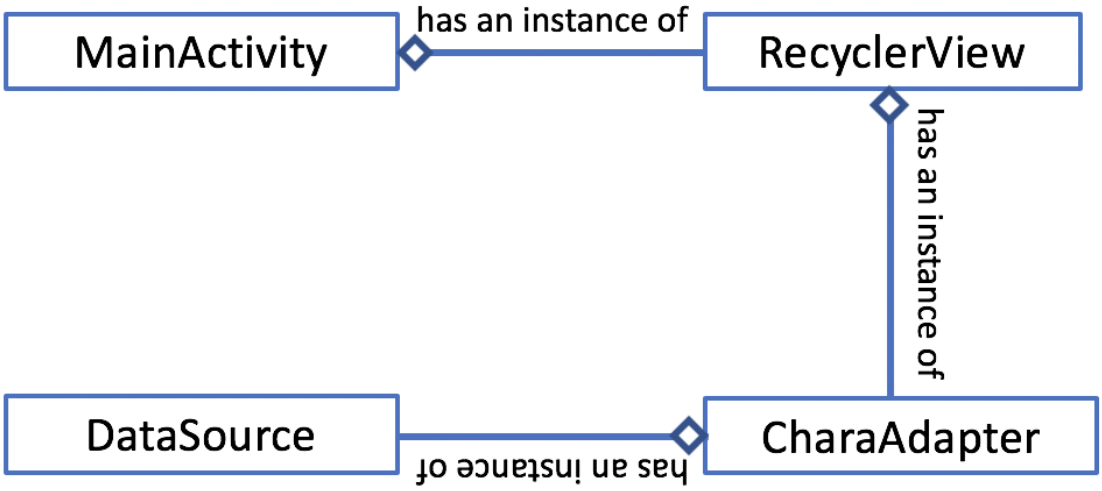
Building the RecyclerView (TODOs 11.x)

Before You Begin

- ensure that the res/drawables folder has a sufficient number of images (six or more)
- Examine the **`DataSource`** class WHERE?
- Examine the **`CharaAdapter`** class
- Examine the **`Utils.firstLoadImages`** method
- Examine **`pokemon.xml`** in res/layout

The relationship between the classes

The diagram below is simplified to show you the main ideas in this section.



TODO 11.1 [MainActivity] Get References to the Widgets.

You should obtain references to the RecyclerView and the ImageView.

TODO 11.2 [MainActivity] Create your datasource object by calling Utils.firstLoadImages

Create an arraylist of ids to the images in drawables. Hints: refer to Lesson 1 exercise
Then pass it to **`Utils.firstLoadImages`**, which returns a DataSource object.

TODO 11.3 [CharaAdapter] Complete the constructor by initializing the dataSource instance variable

Hints: findViewById() is an instance method of View class

TODO 11.4 [CharaAdapter] Go to CharaViewHolder inner class and complete the constructor to initialize the instance variables

This is an inner class representing each instance of the CardView on the RecyclerView.

TODO 11.5 [CharaAdapter] The layout of each CardView is inflated and used to instantiate CharaViewHolder. No coding needed.

There are three methods that you must override for a RecyclerView adapter.
`onCreateViewHolder` is the first. The code has been written for you.

TODO 11.6 [CharaAdapter] The data at position i on the dataSource is extracted and given to the i-th card.

Next, complete **`onBindViewHolder`** to get the data from dataSource and assign it to the widgets in each card.

TODO 11.7 [CharaAdapter] Return the total amount of data points.

Lastly, complete **`getItemCount`** to get the number of data points. Android depends on this method to populate the RecyclerView. If this returns 0, there will be no Cards in the RecyclerView.

TODO 11.8 [MainActivity] Complete the necessary code to initialize the RecyclerView

- Instantiate **`CharaAdapter`**
- Assign the **`CharaAdapter`** instance to the **`RecyclerView`** widget
- Assign a **`LayoutManager`** instance to the **`RecyclerView`** widget. You can choose between **`LinearLayoutManager()`** and **`GridLayoutManager()`**

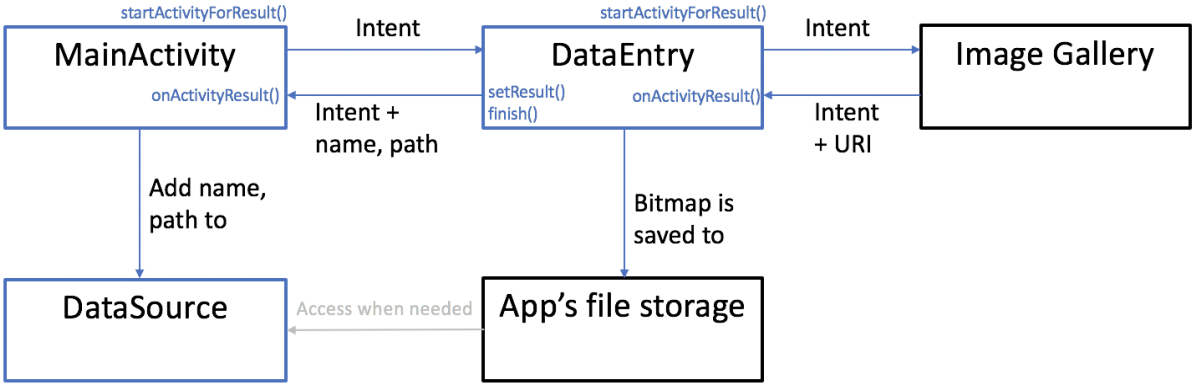
Getting Images From The Image Gallery

Before You Begin

- If you are using the emulator, do put a few images in the image gallery (please find out yourself how to do it)
- Examine the **`DataEntry`** class
- Examine **`activity_data_entry.xml`** in res/layout

The relationship between the classes

This diagram shows the sequence of events in this app.



The data storage solution that I will be using

This is not ideal, but I would consider it adequate for this app.

- The data in DataSource will be converted to a JSON string and stored using SharedPreferences.
- Images will be stored in the app's internal file storage.

TODO 12.1 [MainActivity] Set up an explicit intent to DataEntry activity with startActivityForResult

- This is done by clicking on the FloatingActionButton.
- The code has been written for you.

TODO 12.2 [DataEntry] Set up an implicit intent to the Image Gallery

- This is done by clicking on the SelectImageButton
- The code has been written for you.

TODO 12.3 [DataEntry] Write onActivityResult to get the image selected

- The URI for the image selected is passed via the Intent.
- The URI is passed to setImageURI
- Get the bitmap image from the URI

When this is completed the image selected from the gallery will be displayed in the DataEntry activity. Some of the code that you need is shown below.

```
Uri fullPhotoUri = data.getData();
bitmap = MediaStore.Images
            .Media.getBitmap(this.getContentResolver(),fullPhotoUri);
imageViewSelected.setImageURI(fullPhotoUri);
```

TODO 12.4 [DataEntry] When the OK button is clicked, set up an intent to go back to MainActivity

- Instantiate the Intent object and get the result code
- Obtain the name from the editTextWidget
- Save the bitmap to internal storage using the Utils.saveToInternalStorage
- Invoke setResult() and finish()

When this is completed, upon clicking OK, you will be brought back to MainActivity. However, the RecyclerView will not be updated with the new image.

```
int resultCode = Activity.RESULT_OK;
Intent resultIntent = new Intent();

//Complete the following code
//Extract the name from the EditText widget:
String name =

//Save the bitmap image to Internal storage:
String path = Utils.saveToInternalStorage( ) ;

//Call putExtra on the intent object to save
//the name and path


setResult(resultCode, resultIntent);
finish();
```

TODO 12.5 [MainActivity] Complete onActivityResult to get the data passed back from DataEntry and add to the DataSource object

- Extract the name and path information from the intent object, **`data`**
- Add the name and path to **`dataSource`**
- Display the newest image on the imageView
- If you'd like, you can display a toast as well
- Inform **`charaAdapter`** that the data has changed by the **`notifyDataSetChanged()`** method

When this is completed, the RecyclerView will be updated with the new image.

TODO 12.6 [MainActivity] Complete onPause to store the DataSource object in SharedPreferences as a JSON string

- Invoke the SharedPreferences.Editor
- Instantiate a GSON object and convert dataSource to a JSON string
- Store this string in sharedPreferences

The code that you will likely need is given below.

```
SharedPreferences.Editor prefsEditor = mPreferences.edit();
Gson gson = new Gson();
String json = gson.toJson(dataSource);
prefsEditor.putString(KEY_DATA, json);
prefsEditor.apply();
```

TODO 12.7 - 12.8 [MainActivity] In onCreate, load the JSON string from shared preferences and initialize your dataSource object

The code to do so is given below.

```
mPreferences = getSharedPreferences(PREF_FILE, MODE_PRIVATE);
Gson gson = new Gson();
String json = mPreferences.getString(KEY_DATA, "");
dataSource = gson.fromJson(json, DataSource.class);
```

One thing to note is that if **`json`** is empty, that means the app is starting for the first time. How would you handle this situation?