

Lesson 4.1 - Pokédex

This is an updated version of Lesson 4 since 2019.

Objectives

- Calling intents using `ActivityResultLauncher` in an android app
- Implement a `RecyclerView` in an Android app
- Describe the use of static nested classes
- Describe the use of the `Iterator` and `Iterable` interfaces
- Describe the use of the `synchronized` keyword
- 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	
To validate that your app behaves as it should when a user interacts with it	
Bring the user from the app to another app with a specific function	
The series of callbacks as an Activity is created and/or destroyed	
To run long tasks in another thread	
Store data to make it available at another point in time	
To validate that code components (e.g. methods) in your app behaves as it should	
To develop reusable code by subtyping polymorphism	
To develop reusable code by parametric polymorphism	
To develop reusable code by subtyping polymorphism and parametric polymorphism	

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  
  
    }  
  
}
```

Recall concurrency

Remember that in Lesson 3, the **ExecutorService** object manages the threads that you wish to have in your program, and the **execute()** method executes a **Runnable** object for tasks to be run in the background. **You have no control over the order of the statements from each thread that are to be executed.** Thus, one possible output of the following is

Q0 P0 Q1 P1 P2 Q2 Q3 P3 Q4 P4 END

END

```
public class TestRunnables {

    public static void main(String[] args){
        // you want two threads and execute two tasks
        ExecutorService executorService=Executors.newFixedThreadPool(2);
        executorService.execute( new PrintStr("Q", 5));
        executorService.execute( new PrintStr("P", 5));
        executorService.shutdown();
    }
}

class PrintStr implements Runnable {

    private String s;
    private int times;

    PrintStr(String s, int times){
        this.s = s;
        this.times = times;
    }

    @Override
    public void run() {
        for(int i = 0; i < times; i++){
            System.out.print(s + i + " ");
            waitFor(10);
        }
        System.out.println("END");
    }

    public static void waitFor(int nanoSeconds){
        try{
            Thread.sleep(0, nanoSeconds);
        }catch( InterruptedException ex){
            ex.printStackTrace();
        }
    }
}
```

If you try to update a common resource from more than one thread, you will get a race condition

A **race condition** occurs when two or more threads are trying to update the same resource at the same time. This leads to errors in the outcome of what you are trying to do.

In the following example, there is one instance of an **Account** object (the shared resource). The **Add** class has a **run()** method, hence it is meant to be executed in the background. It takes an **Account** object and, in the **run()** method, deposits 1 to the **Account** object's balance by calling the **deposit()** method.

To illustrate a race condition, we break up the code in **deposit()** to two statements:

- Calculate the new balance: **int newBalance = this.balance + amount;**
- Update instance variable balance with the new value: **this.balance = newBalance;**

Pause for 10 nanoseconds between the two statements to make the wrong outcome clearer.

(Note: if there was only one statement in **deposit()**, **this.balance += amount;** you would not see a race condition).

To illustrate what could go wrong, the two threads could execute the statements in the following order:

Step	Thread 1	Thread 2
1	int newBalance = this.balance + amount;	
2		int newBalance = this.balance + amount;
3		this.balance = newBalance;
4	this.balance = newBalance;	

You can see that this does not result in the correct balance.

We execute 100 threads to run 100 instances of this **Add** class in the background. You would normally expect the outcome of **getBalance()** to be 100, but because of the race condition, you will get a lower value.

Acknowledgements. This example and the code on the next page is taken from Y. Daniel Liang, "Introduction to Java Programming", 10th edition, Chapter 30, Section 30.7.

```

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class TestExecutor {

    public static void main(String[] args) {
        Account account = new Account();

        ExecutorService executorService=Executors.newCachedThreadPool();
        for( int i = 0; i < 100; i++){
            executorService.execute( new Add(account));
        }
        executorService.shutdown();
        while( !executorService.isTerminated()){

        }

        System.out.println(account.getBalance()); //does not display 100!
    }

    private static class Add implements Runnable{

        private Account account;

        Add( Account account){
            this.account = account;
        }
        @Override
        public void run() {
            this.account.deposit(1);
        }
    }

    private static class Account{

        private int balance;

        public int getBalance() {
            return balance;
        }
        public void deposit(int amount){
            int newBalance = balance + amount;
            waitFor(1);
            balance = newBalance;
        }
    }

    public static void waitFor(int nanoSeconds){
        try{
            Thread.sleep(0, nanoSeconds);
        }catch( InterruptedException ex){
        }
    }
}

```


To avoid more than one thread entering a critical region, use the **synchronized** keyword

A **race condition** occurs when two or more threads are trying to update the same resource at the same time.

The critical region is where the shared resource is updated. In the previous example, it would be the **deposit()** method in the account class.

To ensure that the **deposit()** method is only executed by one thread at a time, put the **synchronized** keyword in the method header. This makes the method **thread-safe**. By doing so, a lock is acquired at the start of the method, and released at the end of the method, so that only one thread can access the statements in between.

```
private static class Account{

    private int balance;

    public int getBalance() {
        return balance;
    }
    public synchronized void deposit(int amount){
        int newBalance = balance + amount;
        waitFor(1);
        balance = newBalance;
    }
}
```

You can see that by this synchronization, the balance is updated correctly.

Step	Thread 1	Thread 2
1	<i>Lock is acquired on deposit()</i>	
2	int newBalance = this.balance + amount;	
3	this.balance = newBalance;	
4	<i>Lock is released</i>	<i>Lock is acquired on deposit()</i>
5		int newBalance = this.balance + amount;
6		this.balance = newBalance;

An Iterator object allows you to loop over and remove elements of a list

Recall that you can write a for-each loop to iterate over the elements of a list. However, you are not allowed to remove any element of a list in a for-each loop. Writing code to do so will cause a run-time error to be thrown.

```
List<String> arrayList = new ArrayList<>();

arrayList.add("apple");
arrayList.add("orange");
arrayList.add("pear");
arrayList.add("durian");

for( String word: arrayList){
    System.out.println(word);
    arrayList.remove(word); // causes a run-time error
}
```

If you need to write a loop to remove elements in a List

- Use the **iterator()** method to ask for an **Iterator<T>** object to be returned.
- Use its methods to loop over a list
 - **hasNext()** - returns true if there are elements remaining, false otherwise
 - **next()** - returns the next element
 - **remove()** - remove the element previously returned by a call to **next()**

Read the Javadocs for more details.

```
List<String> arrayList = new ArrayList<>();

arrayList.add("apple");
arrayList.add("orange");
arrayList.add("pear");
arrayList.add("durian");

Iterator<String> stringIterator = arrayList.iterator();

while(stringIterator.hasNext()){
    String myString = stringIterator.next();
    if( myString.equals("pear") ) stringIterator.remove();
}
```

Your class can implement the `Iterable` interface to return an `Iterator` object to avoid exposing the internal representation of your class

Let's say you want to write a class to store a list of strings. Your choice of the internal storage (i.e. private instance variable) is currently an **`ArrayList<String>`**.

You decide that you do want to

- give your clients the ability to loop over and modify the words stored,
- yet you want to hide how the words are stored internally.

The solution is to

- Have your class implement the **`Iterable<T>`** interface from the Java library. This forces you to implement a public method **`public Iterator<T> iterator()`** method that returns an **`Iterator<T>`** object.
- In the example below, since your internal storage is an **`ArrayList`**, you can simply return its **`Iterator`**.

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class SomeList implements Iterable<String> {

    private List<String> stringList;

    SomeList(){
        stringList = new ArrayList<>();
    }

    public void addWord(String word){
        stringList.add(word);
    }

    @Override
    public Iterator<String> iterator() {
        return stringList.iterator();
    }

}
```

If your class is `Iterable`, you can write a for-each loop over the elements of your class, or you can use the `Iterator`

If your class is **Iterable**, you can either (1) write a for-each loop over the elements of your class, or (2) you can use the **Iterator**. The client of your class does not need to know how the elements are stored, and you as the implementer have the freedom to change the internal implementation without breaking the code that your client has written.

```
SomeList someList = new SomeList();
someList.addWord("apple");
someList.addWord("orange");
someList.addWord("pear");
someList.addWord("durian");

// for-each loop
for(String s: someList ){
    System.out.println(s);
}

// Iterator
Iterator<String> stringIterator = someList.iterator();
while(stringIterator.hasNext()){
    System.out.println( stringIterator.next());
}
```

Intents - getting a result

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 another 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()`.

In some scenarios of implicit intents, you would like to get a result after you finished using the other app, e.g. picking an image from a gallery to be displayed in your own app. How do we do that?

Implicit Intent - getting an image from the gallery (2020 update)

Since 2020, Android has a new framework for implicit intents that returns a result. In this section, we will illustrate how this is done by allowing a user to choose an image from the image gallery.

1. User carries out an action e.g. clicks a button to access the image gallery.
2. The result of the user's choice is returned to the app.

Step 1. Specify what you want to do with the result of the user's choice. Create a **launcher**, which is an instance of the **ActivityResultLauncher<Intent>** class. You do this by calling the **registerForActivityResult()** method.

The **launcher** contains code which handles the result from the image gallery e.g. displaying it to a widget. **registerForActivityResult()** takes in two objects.

1. The first object is an instance of the **ActivityResultContract** class. To keep things simple, just remember that you can always use **new ActivityResultContract.StartActivityForResult()**.
2. The second object is an object that implements the **ActivityResultCallback<ActivityResult>** interface, usually as an anonymous class.

The code which handles the result from the image gallery is implemented in the callback **onActivityResult()**. In the example below, the URI of the image retrieved from the image gallery is passed to an **imageView** widget.

```
final ActivityResultLauncher<Intent> launcher = registerForActivityResult(
    new ActivityResultContracts.StartActivityForResult(),
    new ActivityResultCallback<ActivityResult>() {
        @Override
        public void onActivityResult(ActivityResult result) {
            if( result.getResultCode() == Activity.RESULT_OK
                && result.getData() != null){
                Uri photoUri = result.getData().getData();
                testImageView.setImageURI(photoUri);
            }
        }
    }
);
```

Step 2. Write code in the user's action to specify the android system component to be activated.

In code for a button click (for example) create an intent object and pass it to the launcher's launch method.

The Intent constructor has two inputs.

1. The first input to the Intent constructor is `Intent.ACTION_PICK`, which is to specify that the user is to select some data.
2. The second input is `MediaStore.Images.Media.EXTERNAL_CONTENT_URI`, which is the URI for the location of external storage (i.e. outside the app) of images.

<https://developer.android.com/reference/android/content/Intent#public-constructors>

Next, the intent object created is passed to the `launch` method of the `launcher`.

```
testButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Intent intent = new Intent( Intent.ACTION_PICK,
        MediaStore.Images.Media.EXTERNAL_CONTENT_URI);
        launcher.launch(intent);
    }
});
```

Further reading for Reference

1. This section is based on this blogpost
<https://stacktips.com/articles/writing-image-picker-using-intent-in-android>
<https://stackoverflow.com/questions/71604808/how-to-upload-and-save-an-image-from-gallery-startactivityforresult-deprecated>
<https://stackoverflow.com/questions/64950311/what-does-mediastore-images-media-external-content-uri-do>
2. The purpose of the [ActivityResultContract](#) is to tell the Android framework the type of system component that you wish to access.
3. The entire process of getting a result from an activity is explained here:
<https://developer.android.com/training/basics/intents/result>
4. Other common intents are found here and pages within.
<https://developer.android.com/guide/components/intents-common>
5. `registerForActivityResult()` is an instance method inherited from two levels above
<https://developer.android.com/reference/androidx/activity/ComponentActivity>

Explicit Intents - getting a result from another activity (2020 update)

You will recall in Lesson 2 that we used intents to move from one activity to another.

In this section, we will describe how this is done with the launcher framework.

Let's assume here that from **MainActivity**, we will launch an intent to **SubActivity**, which will provide a result to be returned to **MainActivity**.

Step 1. In MainActivity, decide what information SubActivity is going to return and write code to specify what you want to do with the result. Then write the launcher in the origin activity. The launcher has the same two inputs as described in the previous section.

In the example below, the code expects the destination activity to return a string.

- A Bundle object is obtained which contains the data returned by SubActivity.
- To retrieve the String, call **getString()** with the **KEY** to retrieve the data that was sent from Step 3.
- The string is displayed on the text view.

Compare this with **Step 3** to see how the data was sent.

```
final ActivityResultLauncher<Intent> launcher =
registerForActivityResult(
    new ActivityResultContracts.StartActivityForResult(),
    new ActivityResultCallback<ActivityResult>() {
        @Override
        public void onActivityResult(ActivityResult result) {
            Bundle b = result.getData().getExtras();
            String s = b.getString(KEY);
            testTextView.setText(s);
        }
    }
);
```


Step 2. In MainActivity, within code for a user action (e.g. button click), declare an Explicit Intent. Next, call `launcher.launch(intent)`. Contrast this with an Explicit Intent taught in Lesson 2.

```
testButtonGo.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Intent intent =new Intent ( MainActivity.this, SubActivity.class);
        launcher.launch(intent);
    }
});
```

Step 3. In SubActivity, within code for a user action, write code to retrieve whatever data you need and call the `putExtra` method to provide the data to be sent back.

In the code snippet below, when the button is clicked:

- The string entered in an EditText widget is retrieved and assigned to variable **s**.
- An **Intent** object is instantiated and **s** is attached to it through the **putExtra** method and a **KEY** variable (this **KEY** is needed to retrieve the data in Step 1).
- The **setResult()** method sends the data back to the MainActivity.
- Calling **finish()** closes the activity. It triggers the `onDestroy()` on the current activity.

Go back to **Step 1** to see how this data is used.

```
subButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        String s = subEditText.getText().toString();
        Log.d(MainActivity.KEY, "sub" + s);
        Intent intent = new Intent();
        intent.putExtra(MainActivity.KEY, s);
        setResult(Activity.RESULT_OK, intent);
        finish();
    }
});
```

Intents - `startActivityForResult()` - **Deprecated Since 2020**

While this has been deprecated since 2020, there are many code examples that use this and you may find it easier to use. Hence the following sections marked with **deprecated** are retained for your reference and will not be tested in exam.

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()` - **Deprecated since 2020**

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 - **Deprecated Since 2020**

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
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
        ...
    }
}

```

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

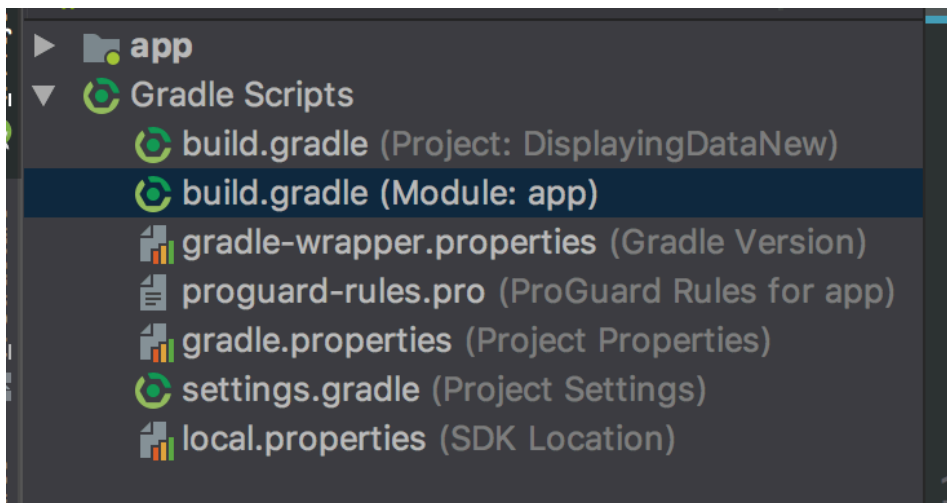
Build → Build APK(s)

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

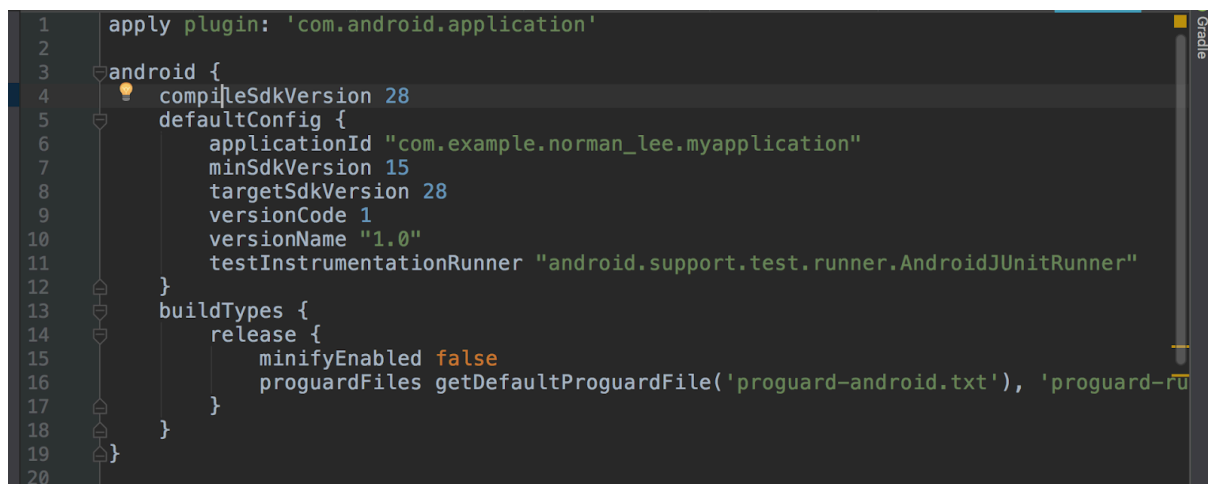
Often the project-level default settings are sufficient.

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 `compileSdkVersion` and `targetSdkVersion` to meet the publishing requirement in Google Play.

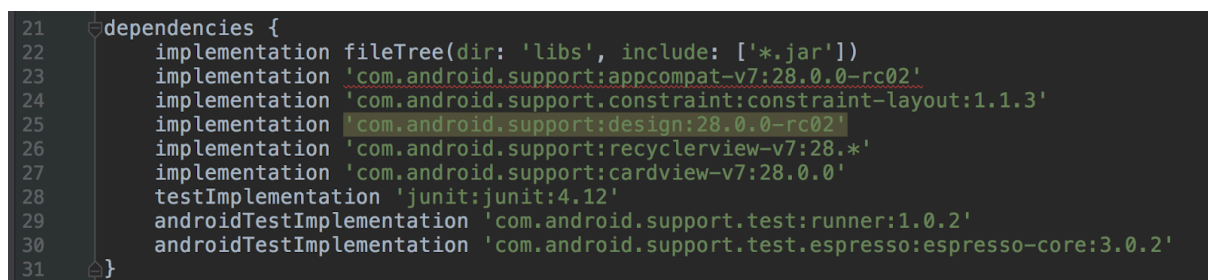


```

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-rules.pro'
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 }
32

```

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

We reiterate the Strategy design pattern here. 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

In doing so, we apply the principle of “**encapsulate what varies**”, so that if there is any change in these behaviours, we just need to create new objects.

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 :(**”

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 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?

We write an adapter class that

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

```
public class TurkeyAdapter implements Duck {

    Turkey turkey;

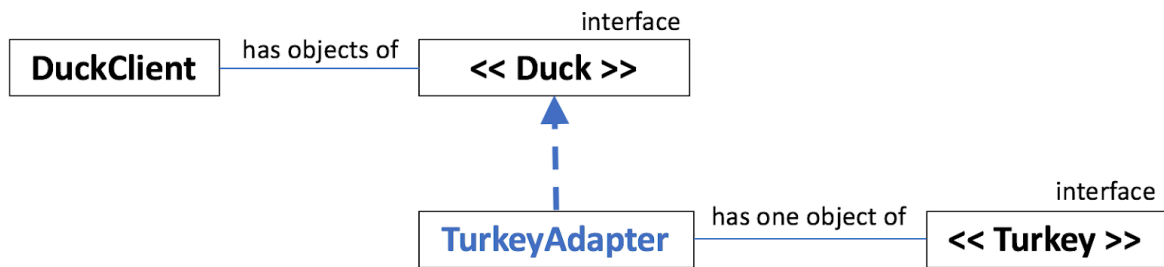
    TurkeyAdapter(Turkey turkey){
        this.turkey = turkey;
    }

    @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.

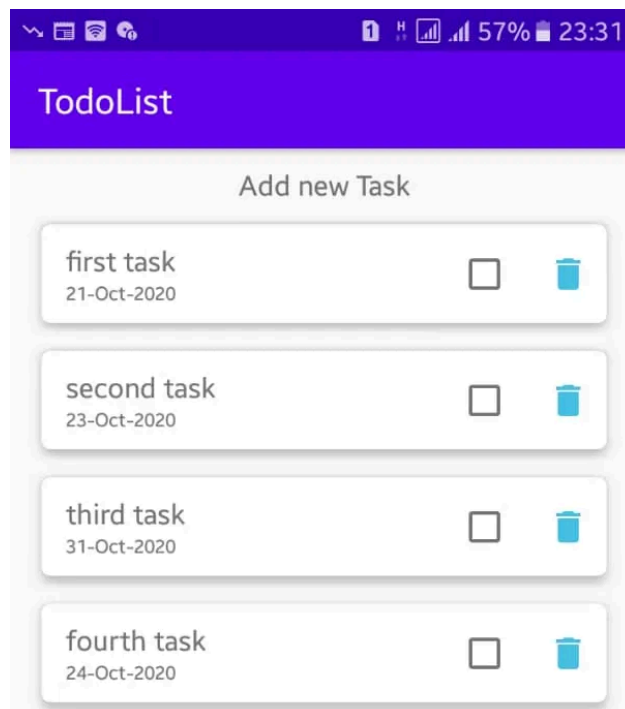
- Images with descriptions
- Chat messages with sender's name

... 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 layout in RecyclerView.

A typical RecyclerView display is shown below. Notice that it looks like a list of identical layouts which the user can scroll up and down, just like many social media apps and e-commerce apps.



This image is adapted and taken from

<https://bamideleomonayin.medium.com/a-simple-explanation-of-how-to-use-recyclerview-on-android-e8aec236b67f>

7f

Why use RecyclerView ?

From the documentation (emphasis mine)

<https://developer.android.com/develop/ui/views/layout/recyclerview>

“RecyclerView makes it easy to efficiently display large sets of data.

You supply the data and define how each item looks, and the RecyclerView library dynamically creates the elements when they're needed.

Thus, imagine that you have a large set of data that has to be displayed as a scrolling list of identical layouts (e.g. Instagram, Shopee etc).

The developer's job is to

- design how each list item layout will look like
- write code to assign data to each list item layout. This is done with an Adapter class.

We will introduce the classes that you have to write in the subsequent pages.

The Android OS then takes care of creating, displaying and hiding each list item as the user scrolls up and down.

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 'androidx.cardview:cardview: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.

Change to the version number recommended by Android studio.

```
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/cardRecyclerView"  
    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.

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 stored locally in your app in various locations.

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.cardRecyclerView);
dataSource = ??? ;
cardAdapter = new cardAdapter(this, dataSource );
recyclerView.setAdapter(cardAdapter);
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 can be declared in the following way.

```
public class CardAdapter extends
RecyclerView.Adapter<CardAdapter.CardViewHolder>{

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

Having designed your CardView layout for each data item, **CardViewHolder** 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 inflater object to be used in **onCreateViewHolder()**.

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

1. 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 is stored in **itemView**
- This reference **itemView** is passed to the constructor of **CardViewHolder**
- **CardViewHolder** uses this reference to get references to the **individual** widgets in the layout

Here's a typical recipe:

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

2. onBindViewHolder() is meant to

- get the i-th data from your data source
- attach it to the widgets through ViewHolder object (i.e CardViewHolder in our example).

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

3. `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

Recall that in the Android framework for the RecyclerView, the developer's job is to

- design how each list layout will look like
- write code to assign data to the layout ← this is done with an Adapter class.

In other words, the developer is responsible for the source of data and how to display each data item. You wrote code for this in the RecyclerView Adapter class.

The Android OS then makes use of the developer's decisions in the RecyclerView adapter class, and takes care of creating, displaying and hiding each list item as the user scrolls up and down.

This is what the Android OS would say to you:

"Dear Developer, you have to design item layout and tell me how you want to put the data on each item layout by writing the Adapter class. Then as the user scrolls up and down, I will take care of the job of creating, displaying and hiding each list item."

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. **CardViewHolder** 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.

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. This allows RecyclerView to update its structure e.g. shifting items when first item is removed.
(Where did the **getBindingAdapterPosition()** method come from?)

```

@Override
public void onSwiped(@NonNull RecyclerView.ViewHolder viewHolder, int i) {
    //code to delete the view goes here
    CardAdapter.CardViewHolder cardVH = (CardAdapter.CardViewHolder) viewHolder;
    int position = cardVH.getBindingAdapterPosition();
    dataSrc.remove(position);
    cardAdapter.notifyItemRemoved(position);
}

```

Building Your App

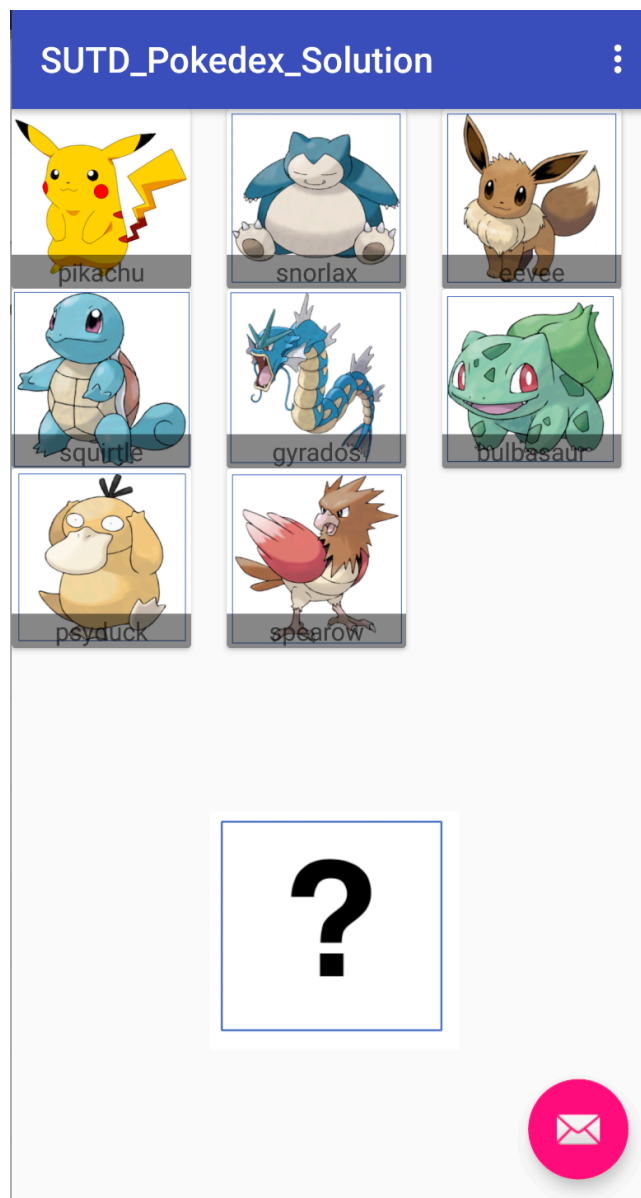
We will first build a RecyclerView using images placed in the drawables folder (TODOs 11.x). As these resources are fixed when your app compiles, we are not able to add images to the RecyclerView.

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**.

The screenshot shows a mobile application interface titled "SUTD_Pokedex_Solution" in a blue header bar. Below the header is a text input field with the placeholder text "enter name". A pink underline is visible under the input field. Below the input field is a grey button labeled "SELECT IMAGE". At the bottom of the screen is another grey button labeled "OK".

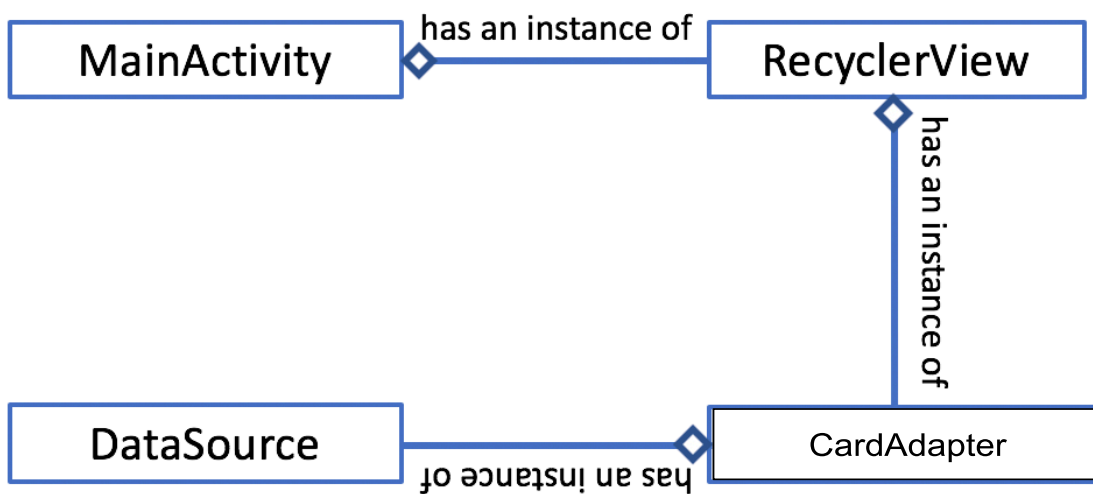
Building the RecyclerView (TODOs 11.x)

Before You Begin

- ensure that the res/drawables folder has a sufficient number of images (six or more)
- Examine the **DataSource** class
- Examine the **CardAdapter** class
- Examine the **Utils** methods
- Examine **card.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] Initialize your datasource from drawable resources using Utils.convertDrawableToCardModel

This code has been written for you

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

TODO 11.4 [CardAdapter] Go to CardViewHolder 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. Use findViewById.

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

There are three methods that you must override for a RecyclerView adapter.

onCreateViewHolder is the first. This code has been written for you.

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

Next, complete **onBindViewHolder** to get the i-th data from dataSource and assign it to the widgets of ViewHolder instance.

TODO 11.7 [CardAdapter] 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 `CardAdapter`
- Assign the `CardAdapter` instance to the `RecyclerView` widget
- Assign a `LayoutManager` instance to the `RecyclerView` widget. You can choose between `LinearLayoutManager()` and `GridLayoutManager()`

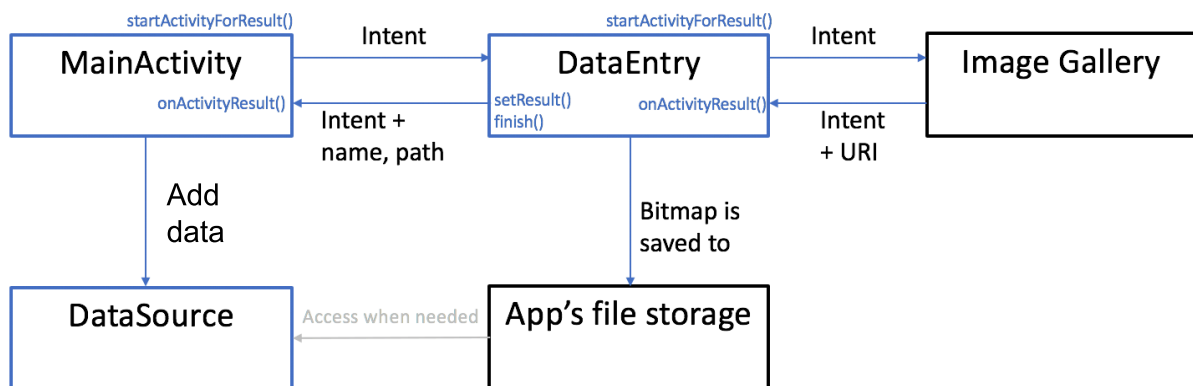
Getting Images From The Image Gallery

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.

You could consider using the Room framework (**deprecated**) to store data in a local SQLite database.

I leave you to explore it for your own purposes.

<https://developer.android.com/codelabs/android-room-with-a-view#0>

If you wish to store the images on the cloud, then one solution is Firebase Storage, which I will leave you to explore.

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

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

TODO 12.3 [DataEntry]

12.3 Write the ActivityResultLauncher to get the selected image

The URI for the image selected is passed via the Intent.

- The URI is passed to setImageURI
- Get the bitmap image from the URI
- This code has been written for you

When this is completed the image selected from the gallery will be displayed in the DataEntry activity.

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()
- This code has been written for you

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.

TODO 12.5 [MainActivity]

12.5 Write the `ActivityResultLauncher` to get the data passed back from `DataEntry` and add to the `DataSource` object

- Extract the name and path information from the intent result. Check the lesson note for the syntax.
- Use the name and path to load image from storage. Hint: use a method from `Utils` class
- Add the new data into **`dataSource`**
- Display the newest image on the `imageView` (which is an image of ? previously)
- If you'd like, you can display a toast as well
- Inform **`cardAdapter`** that the data has changed by the **`notifyItemInserted()`** 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`
- This code has been written for you

The code that you will likely need is given below.

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

- This code has been written for you

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? Ans: This is done in TODO 11.2