CMU 04-801: Advanced Mobile Application Development
Week 2: Recycler Views

Lists

Android has a few different ways to display elements in a list or grid.

- ListView displays a scrollable, vertical collection of data and has been around since API1
  - Simple but not efficient
  - Only supports a layout for a vertical scrolling list
- GridView provided a more flexible layout than ListView
- RecyclerView is a more advanced and flexible version of ListView and GridView and is now recommended (added in API 21 and is compatible back to API 7 through the support library)
  - More flexible
  - More efficient
  - Supports both lists and grids
  - Integrates animations for adding, updating and removing items
  - More complex
  - No OnItemClickListener

RecyclerView

https://developer.android.com/guide/topics/ui/layout/recyclerview

RecyclerView is the container used to display a scrolling list of elements based on large data sets or data that frequently changes.

ViewHolder

The views in the list are represented by view holder objects which are instances of a class that subclasses RecyclerView.ViewHolder. Each view holder is in charge of displaying a single item with a view. The RecyclerView creates only as many view holders as are needed to display the on-screen portion of the dynamic content, plus a few extra. As the user scrolls through the list, the RecyclerView takes the off-screen views and rebinds them to the data which is scrolling onto the screen. The view holder objects are managed by an adapter.

Adapter

Adapters bind data to a layout by providing a common interface into different kinds and sources of data. Android adapters are built to feed data from all sorts of sources(such as an array or a database query) and converts each entry into a view that can be added to a layout. Android has many different types of adapters.

RecyclerView.ViewHolder objects are managed by an adapter that subclasses RecyclerView.Adapter. The adapter creates view holders as needed. The adapter also binds the view holders to their data.

LayoutManager

A LayoutManager is responsible for measuring and positioning item views within a RecyclerView and determining when to reuse item views that are no longer visible to the user. To reuse (or *recycle*) a view, a layout manager may ask the adapter to replace the contents of the view with a different element from the dataset.

- LinearLayoutManager supports both horizontal and vertical scroll lists. This is the most commonly used layout manager with RecyclerView.
- StaggeredGridLayoutManager creates staggered lists similar to the Pinterest layout
- GridLayoutManager is used to display grid layouts

You can also create your own custom layout manager.

RecyclerView does not show a divider between items by default. This was probably done to allow for customization. If you wish to add a divider between items, you may need to do a custom implementation by using RecyclerView.ItemDecoration class.

The RecyclerView model does a lot of optimization work so you don't have to:

- When the list is first populated, it creates and binds some view holders on either side of the list. For example, if the view is displaying list positions 0 through 9, the RecyclerView creates and binds those view holders, and might also create and bind the view holder for position 10. That way, if the user scrolls the list, the next element is ready to display.
- As the user scrolls the list, the RecyclerView creates new view holders as necessary. It also saves the view holders which have scrolled off-screen, so they can be reused. If the user switches the direction they were scrolling, the view holders which were scrolled off the screen can be brought right back. On the other hand, if the user keeps scrolling in the same direction, the view holders which have been off-screen the longest can be re-bound to new data. The view holder does not need to be created or have its view inflated; instead, the app just updates the view's contents to match the new item it was bound to.
- When changes are made to the data you can notify the adapter by calling an appropriate RecyclerView.Adapter.notify…() method. The adapter's built-in code then rebinds just the affected items.

ItemAnimator

Whenever changes are made to the data(add, delete, move), the RecyclerView uses an *animator* to change the item's appearance. This animator is an object that extends the abstract RecyclerView.ItemAnimator class. By default, the RecyclerView uses the DefaultItemAnimator class to provide the animation. If you want to provide custom animations, you can define your own animator object by extending RecyclerView.ItemAnimator.

Tulips

Create a new project called Tulips
Empty Activity template
Package name: the fully qualified name for the project
Language: Java
Minimum SDK: API 21 (21 is the minimum API for Material Design)
Leave check boxes unchecked
(note that the process has been simplified and you don't get to chose the activity name or backwards compatibility)

Add Material Components into the app's grade file

```
implementation 'com.google.android.material:material:1.1.0-rc01'
```

Add colors, create a theme with a Material Components parent theme, and apply it to your app.

Images

Drag the 5 tulip images into the res/drawable folder (or copy/paste)
These are now available through the R class that Android automatically creates.
R.drawable.*imagename*

Tulip List Layout
activity _main.xml
Make sure Autoconnect(magnet) is turned on.
Remove the textview
Add an ImageView to the top of the view and chose the bulbs image.
```
android:src="@drawable/bulbs"
```

Add any missing constraints. If you want it with no gap/border add
```
android:adjustViewBounds="true"
```

Having a content description for an image is optional but makes your app more accessible.
In your strings.xml add
```
<string name="bulbs">Bulbs</string>
```
In activity_main.xml use it for your contentDescription
```
android:contentDescription="@string/bulbs"
```

In the layout file in design view drag out a RecyclerView to display the list(can be found in common or containers) below the image.
Add Project Dependency to add the support library with the RecyclerView. This will add it to your build.gradle(module: app) file.

```
implementation 'androidx.recyclerview:recyclerview:1.1.0'
```

Give the RecyclerView an id `android:id="@+id/recyclerView"`
Add start, end, top, and bottom constraints of 0.
Make sure the RecyclerView has layout_width and layout_height set to “match_constraint” (0dp).

List Item Layout
We also need a layout file to determine what each row in our list will look like. Ours will be super simple with just a TextView.
File | New | Layout resource file
File name: list_item
Root element: android.support.constraint.ConstraintLayout
```
androidx.constraintlayout.widget.ConstraintLayout
```

Source set: main
Directory name: layout

Change the constraint layout’s height to “wrap_content”. If the height is match_constraint each text view will have the height of a whole screen.
```
android:layout_height="wrap_content"
```

Drag out a TextView and make sure it has an id (textView)
Make sure the TextView has layout_width and layout_height set to “wrap_content”.
I also added some top and bottom padding so the text is in the vertical center of the row, some start padding, and made the text larger.
```
android:textAppearance="@style/TextAppearance.MaterialComponents.Headline6"
android:paddingBottom="5dp"
android:paddingTop="5dp"
android:paddingStart="10dp"
```

Once you don't need the default text to help with layout, remove it.

Java class
We're going to create a custom Java class to represent the bulb data we'll be using.
First create a new package for our model. In the java folder select the tulips folder (not androidTest or test)
File | New | Package
Name: model

Then select the new model package and add a new class called Bulb.
File | New | Java class
Name: Bulb
Kind: Class

The Bulb class will only have two data members to store the bulb name and image resource id. We'll have a constructor that takes both members as arguments. We'll also need getter and setter methods for both data members and a private utility method that choses the bulb.
For the constructor, getter, and setter methods Android Studio can generate the code for you.
Code | Generate | Constructor (pick whichever data members you want as arguments, we want both)
Code | Getter and Setter

```
public class Bulb {
    private String name;
    private int imageResourceID;

    public Bulb(String name, int imageResourceID) {
        this.name = name;
        this.imageResourceID = imageResourceID;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getImageResourceID() {
        return imageResourceID;
    }

    public void setImageResourceID(int imageResourceID) {
        this.imageResourceID = imageResourceID;
    }
}
```

Now create another package in tulips called sample and then a Java class in sample called SampleData where we will load our data.
File | New | Package

Name: sample

Then select the new model package and add a new class called Bulb.
File | New | Java class
Name: SampleData
Kind: Class

```
public class SampleData {
    public static List<Bulb> tulipList;

    static {
        tulipList = new ArrayList<>();
        addItem(new Bulb("Daydream", R.drawable.daydream));
        addItem(new Bulb("Apeldoorn Elite", R.drawable.apeldoorn));
        addItem(new Bulb("Banja Luka", R.drawable.banjaluka));
        addItem(new Bulb("Burning Heart", R.drawable.burningheart));
        addItem(new Bulb("Cape Cod", R.drawable.capecod));
    }

    private static void addItem(Bulb item){
        tulipList.add(item);
    }
}
```

In Java the List interface can be implemented as an ArrayList or a LinkedList, we're implementing it as an ArrayList(usually better performing). Defining it as a List is more flexible than defining it as an ArrayList and is recommended.
The static keyword means the data member belongs to the class instead of a specific instance. Only one instance of a static field exists even if you create a million instances of the class or you don't create any. So they're essentially global variables shared by all instances.
In our static initializer we initialize our tulipList as an ArrayList and then use our addItem() method to add instances of our Bulb class to our ArrayList.

Adapter
Now we need to create an adapter which will be used to populate the data into the RecyclerView. The adapter's role is to convert an object at a given position into a list row item to be inserted in the list.

In the java folder select the tulips folder and create a new class for our adapter.
File | New | Class
Name: BulbAdapter
Superclass: androidx.recyclerview.widget.RecyclerView.Adapter

With a RecyclerView the adapter requires the existence of a ViewHolder object which describes and provides access to all the views within each item row.
To indicate that our class will be implementing the ViewHolder class add it to the class definition.

```
public class BulbAdapter extends
RecyclerView.Adapter<BulbAdapter.ViewHolder>
```

You will give you errors until you implement the required methods.
Select the light bulb and choose implement methods to get rid of the error.
Every adapter has three required methods that we need to implement.

- onCreateViewHolder() creates a new ViewHolder object whenever the RecyclerView needs a new one. This is the moment when the row layout is inflated, passed to the ViewHolder object and each child view can be found and stored.
- onBindViewHolder() takes the ViewHolder object and sets the list data for the particular row of the views
- getItemCount() returns the number of items in the list

You will still have an error because we need to define the ViewHolder class which will set up the bindings to the view in the layout file.
In our BulbAdapter class we'll create a ViewHolder class to render the item and a default constructor.

```
public class BulbAdapter extends
RecyclerView.Adapter<BulbAdapter.ViewHolder> {
    class ViewHolder extends RecyclerView.ViewHolder {
        TextView bulbTextView;

      //constructor method
      public ViewHolder(@NonNull View itemView) {
          super(itemView);
          bulbTextView = itemView.findViewById(R.id.textView);
      }
}
```

This should have fixed the errors.
With the viewholder defined, let's set up the BulbAdapter class. We'll define a list of bulbs, Context and a constructor that takes both. We'll use this later in MainActivity to define an instance of this class.

```
private List<Bulb> mBulbs;
private Context mContext;

public BulbAdapter(List<Bulb> mBulbs, Context mContext) {
    this.mBulbs = mBulbs;
    this.mContext = mContext;
}
```

Now we'll implement the 3 required methods.
OnCreateViewHolder() is called automatically by the adapter each time it needs to display a data item. When you inflate the xml layout file, you get a view, and then you wrap that in an instance of your ViewHolder class and return that object.

```
@NonNull
@Override
public BulbAdapter.ViewHolder onCreateViewHolder(@NonNull ViewGroup parent,
int viewType) {
    LayoutInflater inflater = LayoutInflater.from(mContext);
    View itemView = inflater.inflate(R.layout.list_item, parent, false);
    ViewHolder viewHolder = new ViewHolder(itemView);
    return viewHolder;
}
```

OnBindViewHolder is called each time the adapter needs to display a new data item. It passes in the reference to the ViewHolder and the position of the data item in the collection. The job of OnBindViewHolder is to take that data object and display its values.

```
@Override
public void onBindViewHolder(@NonNull BulbAdapter.ViewHolder holder, int
position) {
    Bulb bulb = mBulbs.get(position);
    holder.bulbTextView.setText(bulb.getName());
}
```

getItemCount() returns the number of items in the collection.

```
    @Override
    public int getItemCount() {
        return mBulbs.size();
    }
```

Go into MainActivity.java.
First I'll get access to the array list of tulips from my SampleData class to use in this class.

With the adapter set up, in onCreate() we need to instantiate an adapter with our sample data.
Then we need to set the adapter to the RecyclerView.

We also need to set a Layout Manager for our RecyclerView instance. A layout manager positions item views inside a RecyclerView and determines when to reuse item views that are no longer visible to the user.

We're using a LinearLayoutManager which shows the data in a simple list — vertically or horizontally (by default vertically).

```
public class MainActivity extends AppCompatActivity {
    List<Bulb> bulbList = SampleData.tulipList;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        //get the recycler view
        RecyclerView recyclerView = findViewById(R.id.recyclerView);

        //divider line between rows
        recyclerView.addItemDecoration(new DividerItemDecoration(this,
LinearLayoutManager.VERTICAL));

        //define an adapter
        BulbAdapter adapter = new BulbAdapter(bulbList, this);

        //assign the adapter to the recycle view
        recyclerView.setAdapter(adapter);
```

```
        //set a layout manager on the recycler view
        recyclerView.setLayoutManager(new LinearLayoutManager(this));
    }
}
```

You should be able to run it and see your list of tulips.

Item click listener

RecyclerView does not have a built in way for attaching click handlers to items.

The best structure is to make the adapter as independent from the activity as possible so it's reusable. We'll use an interface to decouple the adapter and the activity.
An interface defines the minimum requirements for one object to usefully talk to another.
When we define an interface, we're saying what the minimum requirements are for one object to talk usefully to another. It means that we'd be able to get the adapter to talk to any kind of activity, so long as that activity implements the interface.

So in our BulbAdapter class we'll create an interface with a method, define an instance of the interface, and pass it to the BulbAdapter constructor.

```
public interface ItemClickListener{
    void onItemClick(int position);
}

private ItemClickListener mItemClickListener;

public BulbAdapter(List<Bulb> mBulbs, Context mContext, ItemClickListener
mItemClickListener) {
    this.mBulbs = mBulbs;
    this.mContext = mContext;
   this.mItemClickListener = mItemClickListener;
}
```

In the ViewHolder class we'll implement View.onClickListener and implement onClick so it calls the method in our interface with the correct list item position.

```
class ViewHolder extends RecyclerView.ViewHolder implements
View.OnClickListener{
    TextView bulbTextView;

    //constructor method
    public ViewHolder(@NonNull View itemView) {
        ….
        bulbTextView.setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
        mItemClickListener.onItemClick(getAdapterPosition());
    }
}
```

In MainActivity we'll implement the interface and override the interface's method to define what should happen when a row is clicked.

```
public class MainActivity extends AppCompatActivity implements
BulbAdapter.ItemClickListener {}

@Override
public void onItemClick(int position) {
    Intent intent = new Intent(MainActivity.this,
DetailActivity.class);
    intent.putExtra("bulb_id", position);
    startActivity(intent);
}
```

DetailActivity will give you an error because we haven't created it yet, we'll do that next.
We use the id to pass which bulb was selected.
When creating the adapter we also need to pass a third parameter now, "this" which refers to the MainActivity.

```
BulbAdapter adapter = new BulbAdapter(Bulb.tulips, this, this);
```

DetailActivity
Now let's create the DetailActivity activity.
New | Activity | Empty
Activity name: DetailActivity
Check Generate Layout File
Package name: com.examples.tulips

Go into the activity_detail layout file and add a TextView for the name. Add a vertical constraint if needed. Remove the default text.

In the TextView make its appearance larger and change the id. You can add text for layout and testing.

```
android:textAppearance="@style/TextAppearance.MaterialComponents.Headline4"
android:id="@+id/textViewName"
```

Add an ImageView for the image. Chose an image for layout and testing purposes then you can remove the src property. If no vertical constraint is added, add one.
Setting all the constraints to match_constraint allows you to set the aspect ratio to 1:1 so the image doesn't get stretched.
By adding a top and bottom (32dp) constraint, along with the aspect ratio, the image will automatically scale and never go off the bottom such as in landscape orientation.
Change the id to `android:id="@+id/imageViewBulb`
It will be asking you for a content description. Add a string resource and then add a content description.

```
<string name="bulb_image">Bulb picture</string>
android:contentDescription="@string/bulb_image"
```

If you run it at this point you should get to that view regardless of which tulip you tap.
Remove the image source since we'll be populating that programmatically.
Now let's populate the view with the correct data.

View Data

In DetailActivity.java we'll get the data sent from the intent to populate the view.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_detail);

    //get bulb id from the intent
    int bulbnum = (Integer)getIntent().getExtras().get("bulb_id");
    Bulb bulb = SampleData.tulipList.get(bulbnum);

    //populate image
    ImageView bulbImage = findViewById(R.id.imageViewBulb);
    bulbImage.setImageResource(bulb.getImageResourceID());

    //populate name
    TextView bulbName = findViewById(R.id.textViewName);
    bulbName.setText(bulb.getName());
}
```

Now when you run it the detail view should show the bulb name and image for whichever bult you clicked on.

It would be nice for the detail view to have up navigation. This is quite simple to add. Just go into the manifests/AndroidManifest.xml file and add a parent activity name for DetailActivity. This will automatically add the up navigation to navigate to the parent.

```
<activity android:name=".DetailActivity"
    android:parentActivityName=".MainActivity">
</activity>
```