Advanced Mobile Application Development
Week 10: Lists and Adapters

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

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)

Images

Drag the 5 tulip images into the 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"

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 the text for this string
<**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.

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
Source set: main
Directory name: layout

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

I also added some bottom padding so the text is in the vertical center of the row and made the text larger.
android:textAppearance="@android:style/TextAppearance.Material.Medium"
android:paddingBottom="10dp"

Java class
We're going to create a custom Java class for the tulip data we'll be using in the next category activity.
In the java folder select the spring folder (not androidTest or test)
File | New | Java class
Name: Bulb
Kind: Class

We're going to create a Bulb class with two data members to store the tulip name and image resource id. We'll have getter and setter methods for both and a private utility method that choses the tulip.

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

```
    //constructor
    private Bulb(String newname, int newID){
        this.name = newname;
        this.imageResourceID = newID;
    }

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

    public String getName(){
        return name;
    }

    public int getImageResourceID(){
        return imageResourceID;
    }

    //the string representation of a tulip is its name
    public String toString(){
        return this.name;
    }
}
```

For now we're putting our tulip array in our data model class and making it public so it can be accessed throughout our activities.
The static keyword means it 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. It will be shared by all instances.

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.

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 position into a list row item to be inserted.

File | New | Activity | Empty
BulbAdapter
Uncheck Generate Layout File
Change the superclass to RecyclerView.Adapter<BulbAdapter.ViewHolder>. This will give you errors until you implement the required methods.
Select the light bulb and choose implement methods to get rid of the error.

If the onCreate() method is giving you an error you can delete it, we won't need it.

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.
So 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);
         }
}
```

Now that we have our adapter and viewholder defined, let's set up the adapter. We'll define a list of bulbs and pass it into the constructor.

```
   private List<Bulb> mBulbs;

   public BulbAdapter(List<Bulb> bulbs){
      mBulbs = bulbs;
   }
```

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

```
   @NonNull
   @Override
   public BulbAdapter.ViewHolder onCreateViewHolder(@NonNull ViewGroup viewGroup, int i) {
      LayoutInflater inflater = LayoutInflater.from(viewGroup.getContext());
      View bulbView = inflater.inflate(R.layout.list_item, viewGroup, false);
      ViewHolder viewHolder = new ViewHolder(bulbView);
      return viewHolder;
   }

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

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

Go into MainActivity.java.
With the adapter set up, we need to bind the data from the adapter to the data source to populate 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 {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

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

        //define an adapter
        BulbAdapter adapter = new BulbAdapter(Bulb.tulips);

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

//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 constructor.

```
public interface ListItemClickListener{
    void onListItemClick(int position);
}

ListItemClickListener itemClickListener;

public BulbAdapter(List<Bulb> bulbs, ListItemClickListener bulbClickListener){
    mBulbs = bulbs;
    itemClickListener = bulbClickListener;
}
```

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) {
        itemClickListener.onListItemClick(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.ListItemClickListener {}

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

BulbActivity 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 second parameter now, “this” which refers to the MainActivity.

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

BulbActivity

Now let’s create the BulbActivity activity.
New | Activity | Empty
Activity name: BulbActivity
Check Generate Layout File
Layout name: activity_bulb
Check Backwards Compatibility and make sure launcher activity is unchecked.

Go into the activity_bulb layout file and add a TextView for the name. Add a vertical constraint if needed.

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

```
android:textAppearance="?android:attr/textAppearanceLarge"
android:id="@+id/bulb_name"
```

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. Setting scaleType to fitCenter scales the image when you rotate the device.
Change the id to `android:id="@+id/bulbImageView"`
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 BulbActivity.java we’ll get the data sent from the intent to populate the view.

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_bulb);
    //get bulb id from the intent
    int bulbnum = (Integer)getIntent().getExtras().get("bulb_id");
    Bulb bulb = Bulb.tulips.get(bulbnum);
    //populate image
    ImageView bulbImage = (ImageView)findViewById(R.id.bulbImageView);
    bulbImage.setImageResource(bulb.getImageResourceID());
    //populate name
    TextView bulbName = (TextView)findViewById(R.id.bulb_name);
    bulbName.setText(bulb.getName());
}
```