Advanced Mobile Application Development
Week 10: 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. RecyclerView recycles elements as they scroll off the screen instead of destroying them so they can be reused for new items. This improves performance and responsiveness of the app. This is done through the View Holder pattern (which is optional in ListView and GridView)

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 RecyclerView binds the view holder to its data through methods in 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 arranging individual elements 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. The RecyclerView library provides three layout managers or you can also create your own custom layout manager.

- 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'll also need to design the layout of the individual items separately.

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
Empty Activity template
Name: Tulips
Package name: the fully qualified name for the project
Language: Kotlin
Minimum SDK: API 21 (21 is the minimum API for Material Design)
Leave legacy libraries unchecked (using these restricts usage of some newer capabilities)
Finish

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

Images

Drag (or copy/paste) the 5 tulip images and the bulb image into the res/drawable folder
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 the image view to have 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.

Give the RecyclerView an id `android:id="@+id/recyclerView"`
Add start, end, top, and bottom constraints of 0. Make sure the top constraint is to the imageView and not parent.
Make sure the RecyclerView has layout_width set to "match_constraint" (0dp) and layout_height is wrap_content.

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: androidx.constraintlayout.widget.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".
Add start and top constraints of 0.
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.
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.

Kotlin class

We're going to create a custom Kotlin 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 Kotlin class called Bulb.
File | New | Kotlin class
Name: Bulb
Kind: Class

The Bulb class will only have two data members to store the bulb name and image resource id.

Remember that a Kotlin class whose main purpose is to hold data should be declared as a data class by using the keyword data.

- The Kotlin compiler will automatically override the toString(), equals(), hashCode(), and copy() methods from the Any class and provide implementations for the data class
    - Derived from the properties declared in the primary constructor
- Eliminates the need for you to manually override these methods and implement them for your data class

Data classes must meet these requirements

- The primary constructor needs to have at least one parameter
- All primary constructor parameters need to be marked as val or var
- Data classes cannot be abstract, open, sealed or inner

Remember that in Kotlin the primary constructor is part of the class header.

```
data class Bulb (val name: String, val imageResourceID: Int){
}
```

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

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

```
object SampleData {
    val tulipList = ArrayList<Bulb>()

    init {
        tulipList.add(Bulb("Daydream", R.drawable.daydream))
        tulipList.add(Bulb("Apeldoorn Elite", R.drawable.apeldoorn))
        tulipList.add(Bulb("Banja Luka", R.drawable.banjaluka))
        tulipList.add(Bulb("Burning Heart", R.drawable.burningheart))
        tulipList.add(Bulb("Cape Cod", R.drawable.capecod))
```

```
    }
}
```

A singleton class is a class that is defined in such a way that only one instance of the class can be created and used everywhere. Kotlin makes it really simple to define a singleton, you just use the keyword "object" instead of "class". We define and initialize an ArrayList with our sample data and because it's in a singleton we'll be able to easily access it from our MainActivity.

Kotlin has both Arrays and Lists. Arrays have a fixed size and are mutable.
Lists can be implemented as an ArrayList or a LinkedList, we're implementing it as an ArrayList (usually better performing).
We define an ArrayList of type Bulb and in the initializer we add our data to the 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 | Kotlin Class
Name: BulbAdapter

The class will need to subclass RecyclerView.Adapter
```
class BulbAdapter: RecyclerView.Adapter {}
```

AS should add the import for the RecyclerView class:
```
import androidx.recyclerview.widget.RecyclerView
```

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.
```
class BulbAdapter: RecyclerView.Adapter<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.

Create this or select the light bulb/more options and choose create class. We could put this class in another file but I just added it in the BulbAdapter class because its behavior is tightly coupled with the adapter.

In our BulbAdapter class we'll create a ViewHolder class to define the layout for an item in our list.

```
class ViewHolder(view: View): RecyclerView.ViewHolder(view) {
    val bulbTextView: TextView = view.findViewById(R.id.textView)
}
```

You will also need to import the class.
```
import com.example.tulips.BulbAdapter.ViewHolder
```

With the viewholder defined, let's set up the BulbAdapter class. We'll define a list of bulbs and add it to the primary constructor.

```
class BulbAdapter(private val bulbList: ArrayList<Bulb>):
RecyclerView.Adapter<ViewHolder>() {} {}
```

Make sure for the 3 required methods that you're using the ViewHolder class and not RecyclerView.ViewHolder.
Now we'll implement the 3 required methods.

1. OnCreateViewHolder() is called automatically by the adapter each time it needs to display a data item.

Create a layout inflater to inflate a view from a XML layout. The context contains information on how to correctly inflate the view. In an adapter for a recycler view, you always pass in the context of the parent view group, which is the RecyclerView.
Then you wrap the view in an instance of your ViewHolder class and return that object.

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
ViewHolder {
    //create an instance of LayoutInflater
    val layoutInflater = LayoutInflater.from(parent.context)
    //inflate the view
    val itemViewHolder = layoutInflater.inflate(R.layout.list_item, parent,
false)
    return ViewHolder(itemViewHolder)
}
```

2. 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 fun onBindViewHolder(holder: ViewHolder, position: Int) {
    //get data at the position
    val bulb = bulbList[position]
    //set the text of the textview to the name
    holder.bulbTextView.text = bulb.name
}
```

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

```
override fun getItemCount() = bulbList.size
```

Now we need to connect our adapter to our RecyclerView so it can get the view holders to create the list.
Go into MainActivity.kt.
First we'll get access to the array list of tulips from my SampleData singleton to use in this class.

```
private var bulbList = SampleData.tulipList
```

In onCreate() we need to
1. set up our recyclerView
2. instantiate an adapter with our sample data
3. set the adapter to the RecyclerView
4. set a Layout Manager for our RecyclerView instance
    a. A layout manager positions item views inside a RecyclerView and determines when to reuse item views that are no longer visible to the user.
    b. We're using a LinearLayoutManager which shows the data in a simple list — vertically or horizontally (by default vertically).

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

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

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

    //define an adapter
    val adapter = BulbAdapter(bulbList)

    //assign the adapter to the recycle view
    recyclerView.adapter = adapter

    //set a layout manager on the recycler view
    recyclerView.layoutManager = LinearLayoutManager(this,
LinearLayoutManager.VERTICAL, false)
}
```

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

Click Handler
RecyclerView does not have a built-in way for attaching click handlers to items so we need to set that up ourselves.

The best structure is to make the adapter as independent from the activity as possible so it's reusable. In our BulbAdapter class we'll add a second parameter in the constructor that represents the click listener. This parameter will actually be a function. This function takes an instance of our class Bulb as a parameter and since it doesn't return anything the return type is Unit. Because this is passed into the class it's available throughout the class.

```
class BulbAdapter(private val bulbList: ArrayList<Bulb>, private val
clickListener:(Bulb) -> Unit): RecyclerView.Adapter<ViewHolder>() {}
```

We want each item in the list to be able to handle click events so we need to set a click event listener to each view holder. The easiest place to do this is in onBindViewHolder() since we have access to the specific item. So now every view holder will have a click event as well.

```
override fun onBindViewHolder(holder: ViewHolder, position: Int) {
    ...
    //assign click listener
    holder.itemView.setOnClickListener{clickListener(bulb)}
}
```

You might notice the syntax of what's being passed into the setOnClickListener() method has curly braces because we're passing in a lambda expression. In Kotlin a lambda expression is similar to a closure in Swift, or an anonymous function in other languages like JavaScript. They don't need a name and are used like an expression. Lambda functions are always surrounded by curly braces.

Now the BulbAdapter and ViewHolder are ready to process click events.
To handle click events on list items we need to define what we want to do when a click event fires.
In MainActivity we'll create a method to define what should happen when an item is clicked. When an item is clicked we want to start a new activity that will show the bulb's name and image.

```
private fun itemClicked(item : Bulb) {
    //create intent
    val intent = Intent(this, DetailActivity::class.java)
    intent.putExtra("name", item.name)
    intent.putExtra("resourceID", item.imageResourceID)

    //start activity
    startActivity(intent)
}
```

DetailActivity will give you an error because we haven't created it yet, we'll create that next.
We create an explicit intent, add the name and resourceID, and then start the intent.
Note that I'm using startActivity() instead of startActivityForResult() since no data is going to be sent back.

The last step in MainActivity is we need to pass this method into the BulbAdapter class as its second parameter. In onCreate() update the line where we define and assign the adapter.

```
val adapter = BulbAdapter(bulbList, {item: Bulb -> itemClicked(item)})
```

This second parameter is also a lambda expression. The parameter item is of type Bulb. The code after the -> is the body, which in this case is the function we just created, and it takes item as a parameter so it knows which item was clicked.

Now let's create the detail activity.

DetailActivity

New | Activity | Empty
Activity name: DetailActivity
Check Generate Layout File
Layout Name: activity_detail
Not a launcher activity
Package name: com.examples.tulips
Language: Kotlin

Go into the activity_detail layout file and add a TextView for the name. Add constraints 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"`
Change the id to `android:id="@+id/textViewName"`

Add an ImageView for the image. Chose an image from sample data for layout and testing purposes. Add constraints if needed.
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 (16dp) 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.
Now let's populate the view with the correct data.

View Data
In DetailActivity.kt we'll get the data sent from the intent to populate the view.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_detail)

    //get intent
    val name = intent.getStringExtra("name")
    val resourceID = intent.getIntExtra("resourceID", -1)

    //update view
    val bulbImage:ImageView = findViewById(R.id.imageViewBulb)
    bulbImage.setImageResource(resourceID)
    val bulbName:TextView = findViewById(R.id.textViewName)
    bulbName.text = name
}
```

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