Week 1 - Foundation

Exploring Android Foundations

CODEPATH

FBU
FACEBOOK UNIVERSITY

Topic Overview

1. **Resources**. App resources and how can we use them?
2. **Organization**. How to organize application source files?
3. **Activities**. How does an activity work and what is a lifecycle?
4. **Layouts**. How do we leverage layouts to build our UIs?
5. **Views**. How do we use and configure common views?
6. **Lists**. How can we build lists of items and make them fast?
7. **Media**. How do we embed images and videos into an activity?

Resources

Understanding Resources

In Android, **Resources** are assets within your app that can be used throughout. This includes:

- **XML Layouts**
- **Values**: Dimensions (`24sp`), Strings, Colors, Styles
- **Drawables / Images**
- **Menu / AppBar Items**
- **Animations**

Understanding Resources

In XML, resources are accessed by a special syntax

@string/my_string - references string in strings.xml
@drawable/cool_image - references image in drawable folder.

In Java, at compile time the resources folders are inspected and a special class called R is generated. The R class can be used anywhere.

```
R.string.my_string
```

Organizing Resources

There are very specific places to put your app resources:

XML Layouts	res/layout/activity_main.xml	@layout/activity_main
Drawable	res/drawable/image.png	@drawable/image
Colors	res/values/colors.xml	@color/red
Dimensions	res/values/dimens.xml	@dimen/title_padding
Strings	res/values/strings.xml	@string/add_button
Styles	res/values/styles.xml	@style/big_blue_button

Creating String Resources

In Android, **Strings** are typically not hard-coded in your application but instead stored in **strings.xml**

- The strings.xml file is used to define a key "name" for the string and the value which is the text.

```
<resources>
    <string name="some_name">My String Text</string>
</resources>
```

Referencing String Resources

You can access any strings defined as:

- "@string/some_name" (**XML**)
- R.string.some_name (**Java**)

Never hardcode any UI strings into your XML or menu layouts. Keep them separate.

Dimension Units

In order to support a variety of screen densities, you should **use relative units** instead of absolute units.

- The most common units within Android development are **dp** (density independent), and **sp** (scale independent).
- Rule of thumb: **sp** for text size, **dp** for everything else.
- Do NOT use px or pt.
- The sp units for fonts will adjust for **both** the **screen density** and user's system **font preference**.

Code Organization

Organizing Android Apps

An android app has a **very specific folder structure**, with all code organized into a particular pattern:

- *src* - This is where all Java source files are located
- res/layout - XML defining view layouts
- res/drawable - Place to store images
- res/values - Strings, colors, etc
- AndroidManifest.xml - Application-wide settings
- build.gradle - Build file (declare dependencies here)

Android Manifest File

AndroidManifest.xml is in every android application and contains application-wide settings. The manifest specifies:

- Package and application name
- Which activity launches on startup
- The components and views of the application
- Permissions that the app requires

Android Manifest: Activity Launcher

- Consider how the first activity interface is displayed once an application is launched.
- This happens as part of the **AndroidManifest.xml**

```
<activity
    android:name="com.example.demoapp.MyActivity"
    <intent-filter>
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

- The Activity that is marked with the **LAUNCHER** category is started when the application first runs.

Gradle Build Files

Gradle is the build system that comes with Android Studio. It's build settings are contained in a build.gradle file. The build file specifies:

- Android specific build options (targetSdkVersion, etc)
- Remote library dependencies
- Version information for the app
- The version of android the app targets

Activities

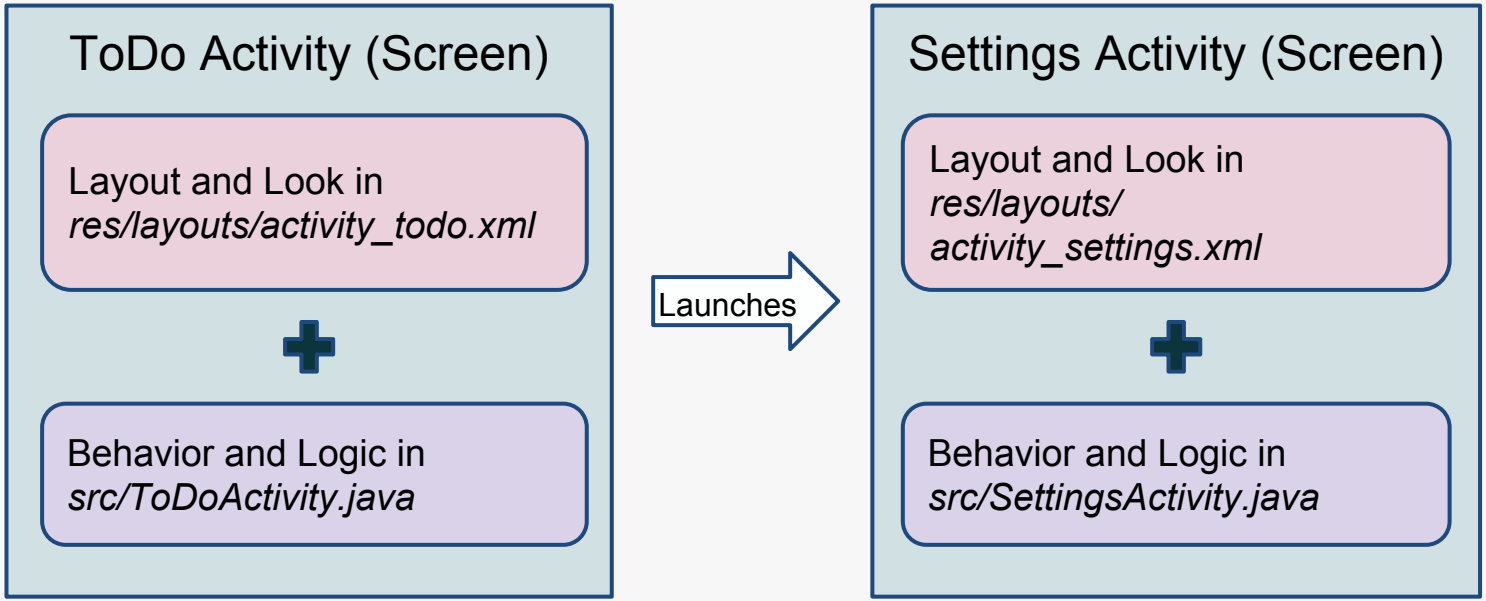
Activity

In Android, each full-screen within an application is called an **Activity**.

- An application can have one or more activities that make up the interaction flow.
- Activities have at least two related files:
  - The Java source file in src/package/FooActivity.java
  - The XML layout in res/layout/foo_activity.xml

Activity

Anatomy of an Android App



Activity

- Activities are each independent and do not **directly** share instances of objects in memory.

- To communicate, activities use a messaging system called **Intents** to send data back and forth.

Activity Lifecycle

Each activity has certain **methods automatically invoked by the Android OS** during initialization, pausing, and destruction.

- onCreate + onResume + onStart
- onPause + onStop + onDestroy

These fire at different times and each are used at different times. See the lifecycle guide for more details.

Activity Lifecycle

An Android activity transitions through various states as it is shown, hidden, and destroyed. Few of them are below:

- **onCreate** - Called to create an activity. Usually sets the xml layout to use as the interface.
- **onPause** - Called when leaving an activity. Usually where any needed data is stored for later.
- **onResume** - Called when returning to an activity. Any stored data is restored here.

Layouts

Activity Layout

- Within an activity screen, the entire user interface is described within a "layout" XML file.
- The XML file generally contains two categories of objects: **Views** and **ViewGroups**.
- The XML file contains the view hierarchy for the screen and should only contain views, layout attributes, and nesting structure.

Activity Layout

- A **View** is any component on screen that is displayed and accepts user interaction such as a textbox or a button.
- A **ViewGroup** is any component that can contain views or other ViewGroups the most common of which is called a **Layout**.
- User interfaces in Android in general involve many **views** displayed in nested **layouts**.

Activity Content Inflation

- When your android application is compiled, every XML layout is accessible as a *resource*.
- The XML layout is usually loaded into an activity during the *OnCreate* lifecycle event using setContentView

```
// Assuming a "res/layout/main_layout.xml"
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main_layout);
}
```

- The process of loading an XML file and turning that into objects in an activity is called **inflation**.

Activity Layout Types

- The XML file that defines the interface for an activity almost always starts with declaring the **root layout**, which defines how views are placed on the screen.
- For example, the layout XML may start with:

```
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
</RelativeLayout>
```

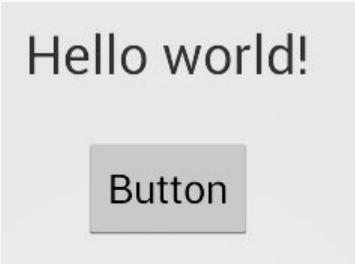
- This declares that within this container, child views that are placed will behave according to the **RelativeLayout** type.

Activity Layout Types

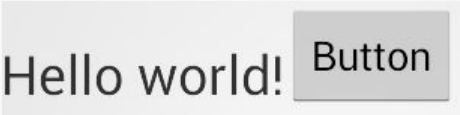
The two most common layouts are **linear** and **relative**.

- **LinearLayout** display views laying out each one after another either horizontally or vertically.

Vertical



Horizontal



Activity Layout Types

- **RelativeLayout** displays views laying out each one based on its relationship with sibling views or relative to the parent.

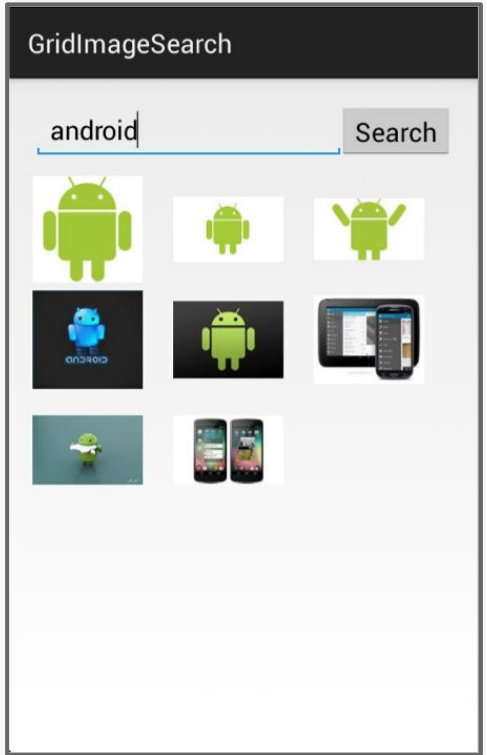
Activity Layout Types

RelativeLayout positions views based on these attributes:

- *Position based on siblings:* **layout_above**, **layout_below**, **layout_toLeftOf**, **layout_toRightOf**
- *Position based on parent:* **layout_alignParentTop**, **layout_alignParentBottom**, **layout_alignParentLeft**, **layout_alignParentRight**
- *Alignment based on siblings*: **layout_alignTop**, **layout_alignBottom**, **layout_alignLeft**, **layout_alignRight**

Activity Layout Types

RelativeLayout positions **based on view relationships**



EditText (etQuery)

```
android:layout_alignParentLeft="true"
android:layout_alignParentTop="true"
android:layout_toLeftOf="@+id/btnSearch"
```

Button (btnSearch)

```
android:layout_alignBottom="@+id/etQuery"
android:layout_alignParentTop="true"
android:layout_alignParentRight="true"
```

GridView (gvResults)

```
android:layout_alignParentLeft="true"
android:layout_alignParentRight="true"
android:layout_below="@+id/etQuery"
```

Layout Parameters

- Every View and ViewGroup is required to specify **layout parameters** that define how that view is placed into the parent layout.
- The two most important are the **layout_width** and **layout_height** parameters.
- While the width and height could be exact measurements, often these are set to special keywords: **match_parent** and **wrap_content**.

Layout Parameters

- In layout parameters, widths or heights are often relative units (dp) or these special keywords.

```
<RelativeLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">
   <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
</RelativeLayout>
```

- This above says that the textview will **fill the width of the layout** but will only be as tall as the content requires.

Activity Layout Types

There are many types of layouts in Android:

- **RelativeLayout** - Children arrange themselves in relation to other views on screen.
- **LinearLayout** - Children are displayed in a linear fashion either horizontally or vertically
- **ConstraintLayout*** - Newer flexible layout that intended to replace RelativeLayout. See CodeLab here.
- **FrameLayout** - Overlaps children on top of each other.
- **GridLayout** - Children are placed in a rectangular grid.

Views

View Margins and Padding

Margins and padding values for views allows us to position and space elements in a layout.

- **Layout Margin** defines the amount of space around the outside of a view
- **Padding** defines the amount of space around the contents or children of a view.

```
<LinearLayout>
   <TextView android:layout_margin="5dp" android:padding="5dp">
   <Button layout_marginBottom="5dp">
</LinearLayout>
```

View Gravity

Gravity and **Layout Gravity** can be used to define the position of the contents of a view.

- **gravity** determines the **position that the contents** of a view will align (like CSS *text-align*).
- **layout_gravity** determines the **position** of the view within it's **parent** (like CSS *float*).

```
<TextView
  android:gravity="left"
  android:layout_gravity="right"
  android:layout_width="165dp"
  android:layout_height="wrap_content"
  android:textSize="12sp" />
```

Basic Views

There are 5 basic view controls that are commonly used to construct user interfaces.

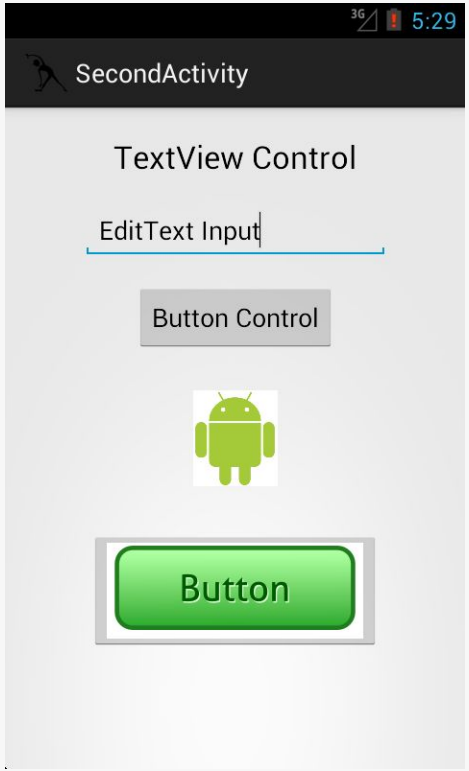
- **TextView** displays a formatted text label
- **EditText** is an editable text field for user input
- **Button** can be clicked to perform an action
- **ImageView** displays an image resource
- **ImageButton** displays a clickable image

Basic Views

There are 4 controls listed in the picture:

- TextView
- EditText
- Button
- ImageView
- ImageButton

All spaced by padding within a LinearLayout.



View Attributes

Every view has many **different attributes** which can be applied to manage various display and behavior properties

- Certain properties are shared across many views such as **android:layout_width**
- Other properties are based on a view's function such as **android:textColor**

```
<TextView
   android:text="@string/hello_world"
   android:background="#000"
   android:textColor="#fff"
   android:layout_centerHorizontal="true" />
```

View Identifiers

Any view can have an **identifier** attached that **uniquely** names that view for later access.

- You can assign a view an id within the xml layout:

```
<LinearLayout>
   <Button android:id="@+id/my_button">
</LinearLayout>
```

- This **id** can then be accessed within the Java code for the corresponding activity (in **onCreate** for example):

```
Button myButton = (Button) findViewById(R.id.my_button);
```

Lists

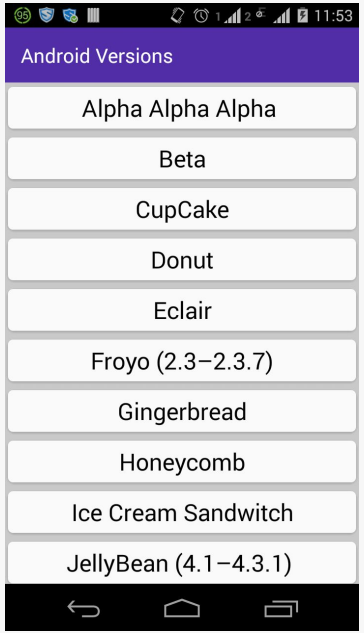
Building Lists with RecyclerView

RecyclerViews display a scrollable list or grid of items from an **Adapter**.

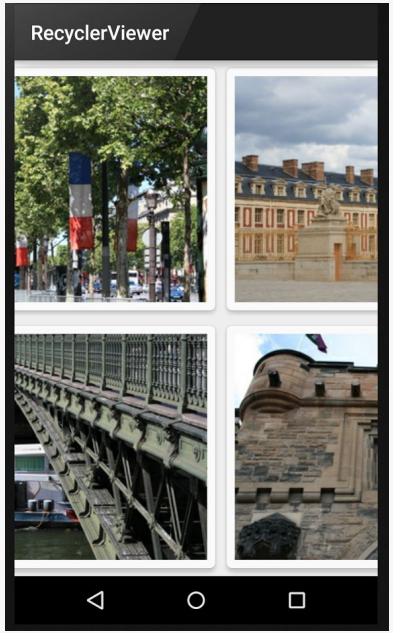
- An adapter automatically fills the items in a RecyclerView **from a source** such as an array or a database query
- Each data item is then **transformed** into a **view item**. The **LayoutManager** helps position the items
- The **ItemAnimator** helps with animating the items (e.g. deletion)

LayoutManager

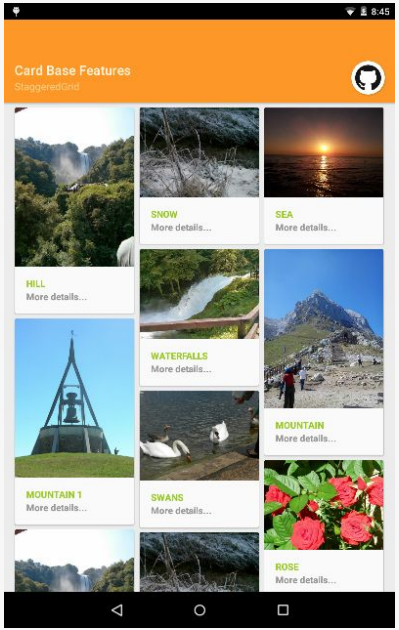
LinearLayoutManager



GridLayoutManager



StaggeredGridLayoutManager



RecyclerView.Adapter

- Handles data collection and binding the items to the view
- Uses the **ViewHolder** pattern

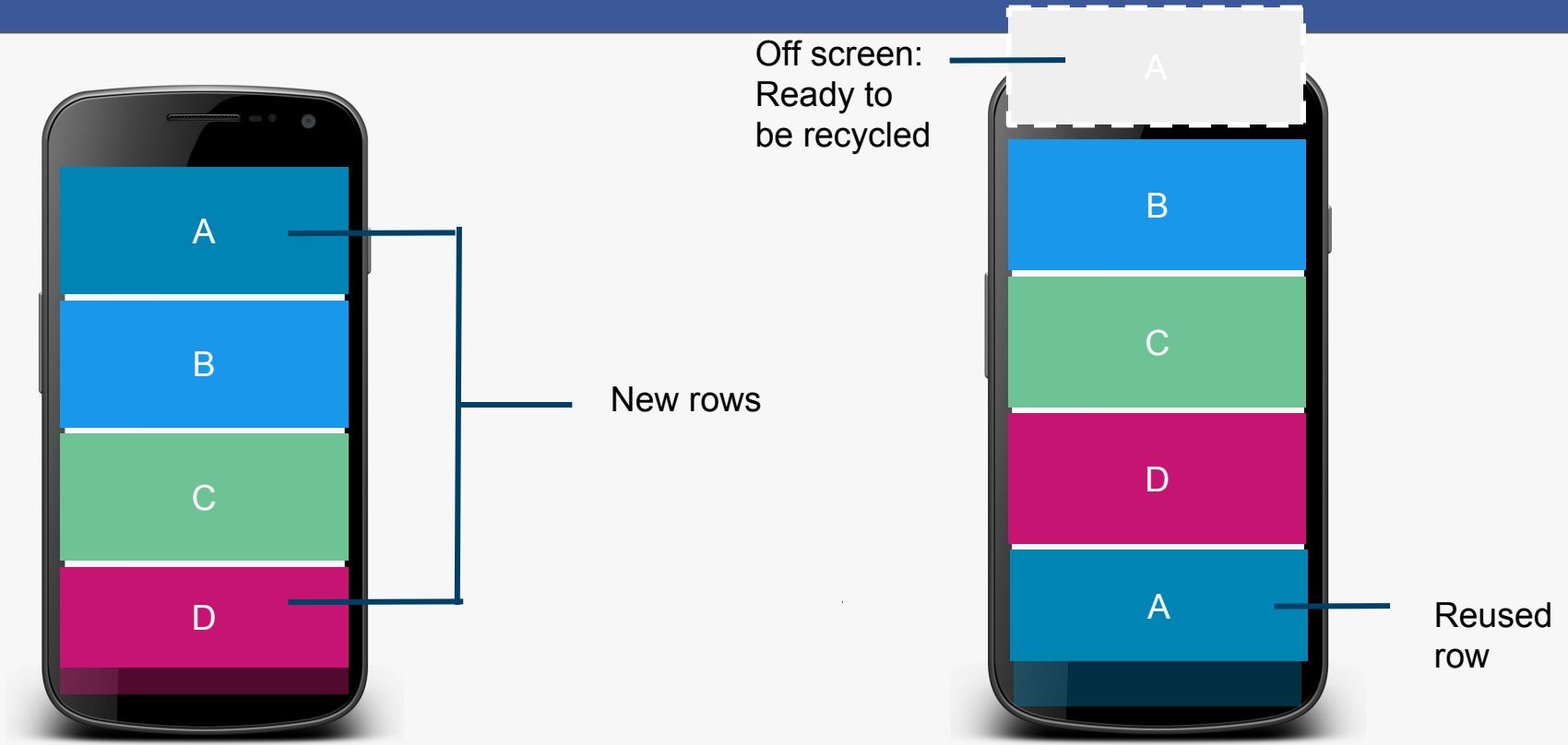
```
public class ContactsAdapter extends  RecyclerView.Adapter<ContactsAdapter.ViewHolder> {
   // ... constructor and member variables
   @Override
   public ContactsAdapter.ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
      // Usually involves inflating a layout from XML and returning the holder
      ViewHolder viewHolder = new ViewHolder(//view inflated from XML);
      return viewHolder;
   }

   @Override
   public void onBindViewHolder(ContactsAdapter.ViewHolder viewHolder, int position) {
      // Involves populating data into the item through holder    }

   @Override
   public int getItemCount() {
       // Returns the total count of items in the list }
}
```

Row View Recycling?

Building **efficient lists in Android** requires smart caching:

- Only **visible rows on screen** are stored in memory.
- As the user scrolls, the same view objects are reused again and again rather than creating new items in memory
- Even in a dataset of 100 items, only 6-7 view objects will be created in memory and be recycled again and again.

Row View Recycling



Notifying the Adapter

- **notifyItemChanged(int pos)**
- **notifyItemInserted(int pos)**
- **notifyItemRemoved(int pos)**
- **notifyDataSetChanged()** *should use as last resort

```
// Add a new contact
contacts.add(0, new Contact("Barney", true));
// Notify the adapter that an item was inserted at position 0
adapter.notifyItemInserted(0);
```

ItemAnimator

- RecyclerView.ItemAnimator will animate changes to the **`ViewGroup`** that are notified to the adapter
  - Add
  - Delete
  - Select
- **`DefaultItemAnimator`** provides basic animations and works quite well
- Use 3rd party libraries for fancier animations

Media

Understanding ImageView

ImageView is simply a view you embed within an XML layout that is used to display an image on screen:

```
<ImageView
    android:id="@+id/image"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:scaleType="center"
    android:src="@drawable/android" />
```

But there’s a lot more complexity than meets the eye.

Understanding ImageView

ImageViews have a lot of hidden complexity and configuration required such as:

- **Scale**. Properly scaling a source image on screen when source dimensions don’t match the ImageView
- **Density.** Ensuring image looks crisp on devices of all resolutions and densities.
- **Memory.** Ensuring that the source bitmap is not too large as to crash the app.

ImageView Gotchas

When working with images and ImageView, remember the following:

- **Icons vs Images.** Don't use the "Image Asset" dialog in Android Studio unless you want to generate small icons.
- **Image Densities.** Use **Final Android Resizer** or other utility to create appropriate images for multiple densities.
- **Memory Errors.** Image files larger than **1776 x 1080px** in actual dimensions might cause Android apps to crash.

ImageView Gotchas, contd

- **Resource Names.** Filenames only include lowercase letters, numbers and underscores (i.e image_1.png)
- **Scaling Images.** Understand and adjust the **scaleType** of your ImageView to control how the image is displayed.
- **Aspect Ratio.** Add android:adjustViewBounds="true" to your ImageView to adjust the size to image aspect ratio.

Image Loading with Glide

When loading an image from a URL, the images need to be downloaded from the network, processed and then resized.

Enter **Glide**, a library that makes loading images from the network incredibly simple.

```
Glide.with(context).load(imageUri)
    .fitCenter()
    .placeholder(R.drawable.user_placeholder)
    .error(R.drawable.user_placeholder_error)
    .into(imageView);
```

Wrap-Up

Topic Overview

1. **Resources**. App resources and how can we use them?
2. **Organization**. How to organize application source files?
3. **Activities**. How does an activity work and what is a lifecycle?
4. **Views**. How do we use and configure common views?
5. **Layouts**. How do we leverage layouts to build our UIs?
6. **Lists**. How can we build lists of items and make them fast?
7. **Media**. How do we embed images into an activity?