Week 1 - Foundations

Exploring Android Foundations

Week 1 - 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.

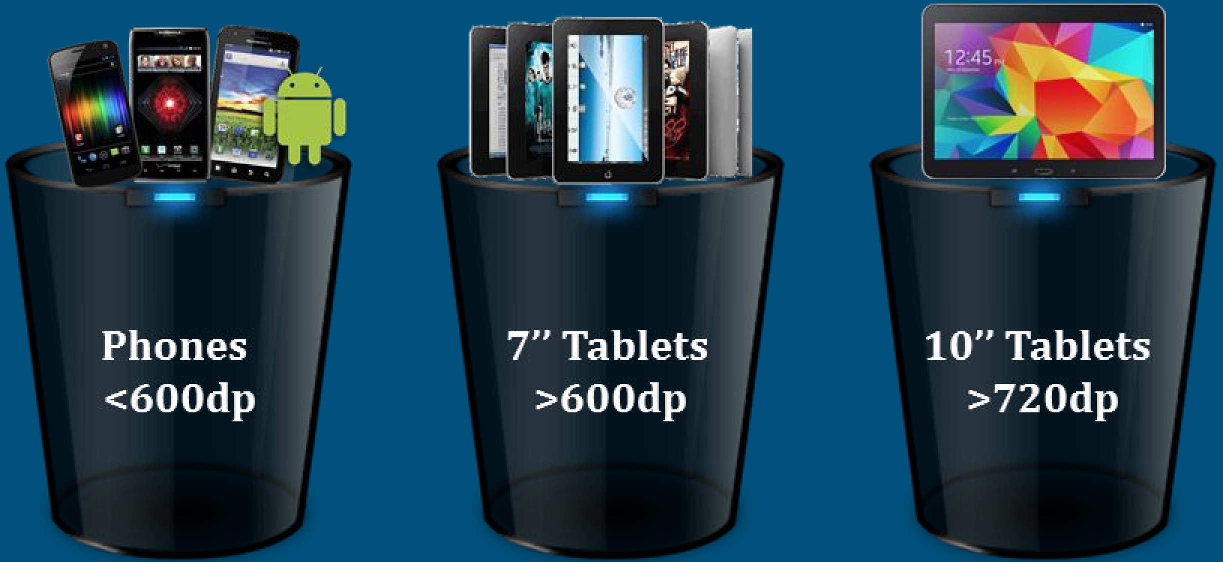
Dimension Units

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**.

Alternative Resources

With Android, there are many different device types and configurations to support:



Alternative Resources

In Android, there are many different device types and situations:

Language	Language code selected on the device	`en, fr`
Screen size	Minimum width of the screen	`sw480dp,sw600dp`
Orientation	Screen is portrait or landscape mode.	`port, land`
Screen density	Pixel density of the screen	`hdpi, xhdpi`

Use alternative resources to make this easy and manageable.

Resources - Discussion Questions

- Why are all the dimension units in **dp** or **sp**? What makes these units important in Android?
- Why are alternative resources so important to building production applications?
- What is the `R` constant and how does this work? What value does each entry have?

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

Better Organizing Android Source Code

Always store your Android source files organized into subpackages by **feature** or **category**.

- **By Category**: Group based on what type of file the source code is (activity, adapter, interface).
- **By Feature**: Group based on the feature the file is contributing towards (details page, creation flow, login)

Organizing Files By Category

- `com.example.myapp.activities` - Activities
- `com.example.myapp.adapters` - Custom adapters
- `com.example.myapp.models` - Data models
- `com.example.myapp.network` - Networking code
- `com.example.myapp.fragments` - Fragments
- `com.example.myapp.utils` - Helpers supporting code.
- `com.example.myapp.interfaces` - Contains all interfaces

Better Separating Concerns

Beyond structuring our sub-packages correctly, we also want to following certain structural best practices:

- Strive for **testable** code. Write code that could be easily unit tested and prefer smaller more focused objects to large files.
- Strive for **maintainable** and **modular** code. Write as little code as possible and prefer simplicity over cleverness.

Better Separating Concerns

- **Keep your activities lean.** The activity should act as a manager and delegator. Avoid having unnecessary logic or code in these files.
- **Keep your models lean.** Models should have fields, validation, JSON parsing, state management and not much else.
- **Use Java service objects.** Create specialized Java classes that perform specific well-scoped functions.

App Architectural Patterns

How are modern production apps organized?

- MVC (Model-View-Controller) - Default Android
- MVVM (Model-View-ViewModel) - Data-binding
- MVP (Model-View-Presenter) - Presenter objects

Code Organization - Discussion Questions

- What is an intent-filter as seen in the Android manifest?
- Is it better to organize code by category or by feature?
- Why do alternate architectures exist such as MVP? Where does the MVC default architecture breakdown?

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 parts:
    - The Java source file in src/package/FooActivity.java
    - The XML layout in res/layout/foo_activity.xml

Activity

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

- Activities are each independent containers and do not **directly** communicate with each other.
- To communicate, activities use a messaging system called **Intents** which we will discuss in more depth soon.

Activity Lifecycle

Each activity has certain **triggers 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);
}
```

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:

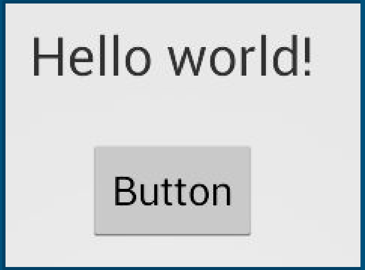
```
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
</RelativeLayout>
```

Activity Layout Types.

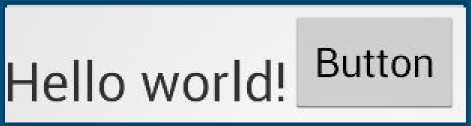
The two most common layouts are **linear** and **relative**.

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

Vertical



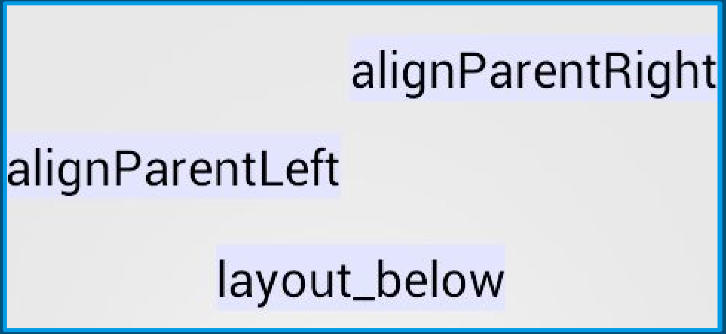
Horizontal



Activity Layout Types, cont'd.

The two most common layouts are **Linear** and **Relative**.

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



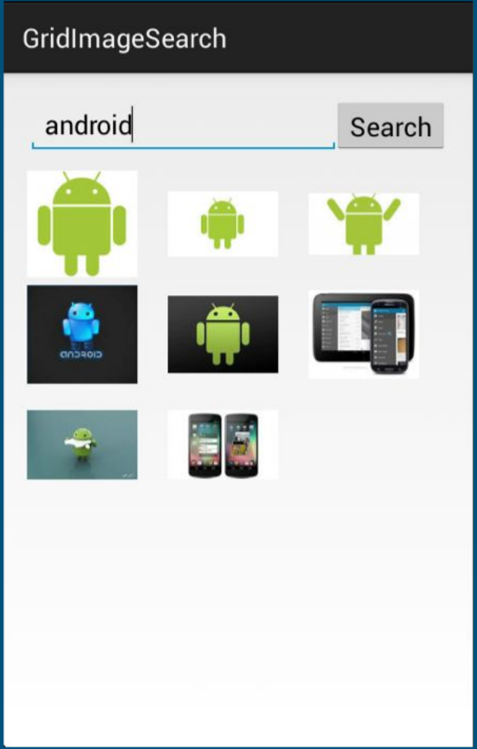
Activity Layout Types, cont'd.

RelativeLayout positions views based on a number of directional 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, cont'd.

RelativeLayout positions based on 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, cont'd

- In layout parameters, widths or heights should be 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 view will **fill the width of the layout** but will only be as tall as the content requires.

Activity Layout Types, cont'd.

There are many types of layouts.

- **LinearLayout** - Children are displayed in a linear fashion either horizontally or vertically
- **RelativeLayout** - Children arrange themselves in relation to other placed children.
- **TableLayout** - Children are arranged into rows and columns.
- **FrameLayout** - Overlaps children on top of each other.
- **GridLayout** - Children are placed in a rectangular grid.

Efficient Layout Optimization

- It is a **common misconception** that using the basic layout structures leads to the most efficient layouts.
- **Minimize the number of instantiated layouts** and especially **minimize deep nested layouts** whenever possible.
- Using **many nested instances of LinearLayout** can lead to an **excessively deep view hierarchy** and can be quite expensive
- A **shallow and wide view hierarchy** is nearly always more efficient to render.

View Layering

- Generally layering / overlapping views is achieved with a **FrameLayout** or **RelativeLayout**.
- Views are drawn in layers by default **based on the order they appear in the XML**.
- In API 21, the `elevation` property was introduced to **explicitly configure the z-index** of your views.
- Learn more in the view layering guide.

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.

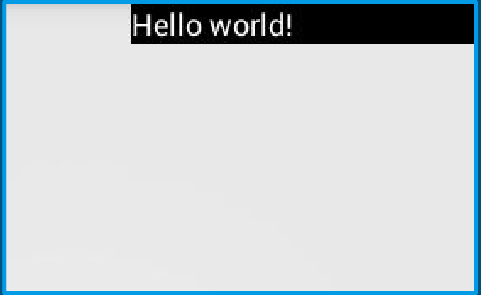
```
<LinearLayout>
   <TextView android:layout_margin="5dp" android:padding="5dp">
   <Button layout_marginBottom="5dp">
</LinearLayout>
```

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" />
```



Hello world!

Basic Views

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

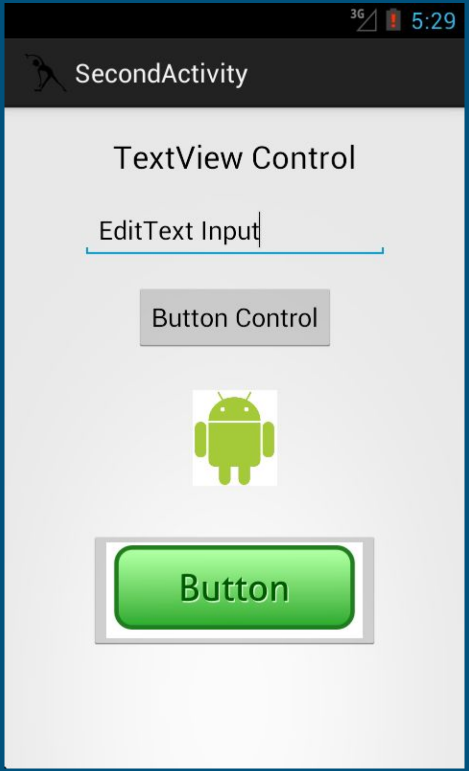
- **TextView** displays a formatted text label
- **EditText** is an editable text field for user input
- **Button** can be clicked to perform an action
- **ImageView** displays an image resource
- **ImageButton** displays a clickable image

Basic Views, cont'd

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

List Views

ListViews display a scrollable list of items from an **Adapter**.

- An adapter automatically fills the items in a ListView **from a source** such as an array or a database query.
- Each data item is then **transformed** into a **view item** within the list.
- There are two common adapters: **ArrayAdapter** and **SimpleCursorAdapter** which can be used as a source of data items for a list.

Array Adapter

ArrayAdapter is the simplest way to fill a ListView given any array of items.

- By default, ArrayAdapter creates a view for each list item by calling toString() on the item and placing the contents in a TextView.
- In example, with an array of strings you want to display:

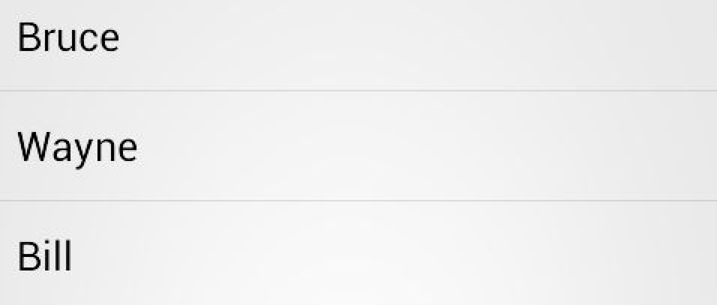
```
String[] nameArray = { "Bruce", "Wayne", "Bill" };
ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
    android.R.layout.simple_list_item_1,
    nameArray);
```

List Views, cont'd

```
List<String> myStringArray = Arrays.asList("Bruce", "Wayne", "Bill");
ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
  android.R.layout.simple_list_item_1, myStringArray);

ListView listView = (ListView) findViewById(R.id.listview);
listView.setAdapter(adapter);
```

becomes a ListView:



Bruce

Wayne

Bill

Array Adapter, cont'd

ArrayAdapter can have data dynamically added or removed from the list.

- Remove all elements with the `clear()` method call.
- Add new elements via the `add()` method
- Add multiple elements with the `addAll()` method.

```
ArrayAdapter adapter = new ArrayAdapter<String>(this,
  android.R.layout.simple_list_item_1, myStringArray);

adapter.clear();
adapter.add("New Item");
adapter.addAll(anotherStringArray);
// even get access to the underlying
adapter.getItem(3);
```

Array Adapter, cont'd

ArrayAdapter can be customized to support complex views using **getView()**. In example, create a custom class from `ArrayAdapter`:

```
public View getView(int position, View convertView, ViewGroup parent) {
    LayoutInflater vi = LayoutInflater.from(getContext());
    View v = vi.inflate(R.layout.complex_item, null);
    ComplexItem item = this.getItem(position);
    TextView tv = (TextView) v.findViewById(R.id.tv);
    TextView tv2 = (TextView) v.findViewById(R.id.tv2);
    tv.setText(item.getName());
    tv2.setText(item.getAddress());
    return v;
}
```

Row View Recycling?

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

- Only **visible rows on screen** are actual views 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.

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, Pt. 1

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** to create appropriate images for multiple densities.
- **Memory Errors.** Image files larger than **1776 x 1080px** in dimensions will cause Android apps to crash.

ImageView Gotchas, Pt. 1

When working with images and ImageView, remember the following:

- **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 Picasso

When loading images from the network, the images need to be downloaded asynchronously, parsed and resized.

Enter Picasso or Glide, libraries that make loading images from the network incredibly easy.

```
Picasso.with(context).load(imageUri)
    .fit().centerCrop()
    .placeholder(R.drawable.user_placeholder)
    .error(R.drawable.user_placeholder_error)
    .into(imageView);
```

Picasso vs Glide

- Syntax is roughly the same between the two
- Glide was created by Google, Picasso by Square
- Both support disk and in-memory image caching
- Glide is almost 3.5 times larger than Picasso in file size of the libraries.
- Glide has 2678 while Picasso has just 849.
- Glide is more memory efficient when loading images than Picasso
- Glide supports playing animated GIFs.

Wrap-Up

Week 1 - 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 and videos into an activity?