alvin alexander

Do not be concerned with the fruits of your action, just give attention to the action itself.
~ Bhagavad Gita

An Android cheat sheet (my notes, main concepts)

By Alvin Alexander. Last updated: February 3, 2024

This page is a little unusual for me; it's basically a terse summary of what I know about Android. I created it because (a) I tend to work with Android for a few weeks or months, and then (b) get away from it for several months, so this page helps me reload everything into my brain.

I don't offer much discussion here; this is mostly just a quick Android reference page. I have written a lot more about Android, and you can follow this link to my Android tutorials, or you can search my website for 'Android'.

Table of Contents

Getting started with Android

- The best way to write Android code today (2017) is to use Android Studio as your IDE
- Android Studio is free, and it's created/maintained by Google
- By far the best book I've found is Android Programming: The Big Nerd Ranch Guide (#ad).
- The Busy Coder's Guide to Android Development is also useful, but it's a little

expensive as a subscription (one of the better things about it is that it offers a historical perspective)

- If you want to create Android games, The Beginner's Guide to Android Game Development (#ad) is a good starter book

Main Android concepts

Some of the main Android concepts to grasp are:

- *AndroidManifest.xml* - describe your app in this file; your app starts with the "main" method you declare here. You also need to declare all of your activities here.
- *Activity* - an `Activity` is a Java controller class that typically corresponds to one screen in your app
- *Fragment* - a `Fragment` is a Java controller that typically corresponds to a widget in a screen (or possibly the full screen)
- You generally design your UI in the designer; this creates XML files that you can also modify as needed
- The Big Nerd Guide has this rule about Activities and Fragments: "always use fragments"
- *Intent* - you launch new activities with Intents
- *Service* - background services, like notifications
- *Content Providers* - tbd
- *BroadcastIntent*, *BroadcastReceiver* - tbd
- `R` class - generated for you by the Android build process

More important concepts

- *View* - widgets like TextView, ImageView, Button ...
- *ViewGroup* - containers for other views
- *Layouts*: FrameLayout, LinearLayout, RelativeLayout, TableLayout, ListView, GridView
- *Menus* (ActionBar, Toolbar)

- *Notifications* - send notifications from your app to the user's tablet or phone; notifications can also be forwarded to Android Wear devices
- Understanding screen densities and sizes
    - dp, sp (and px, in, mm, pt)
    - ldpi, mdpi, hdpi, xhdpi, xxhdpi
    - good Android ui/designer cheat sheet: petrnohejl.github.io/Android-Cheatsheet-For-Graphic-Designers/
- Android.com design principles
- Android UI patterns
- Handling device rotation
- mobile-patterns.com (general mobile ui stuff)

Even more Android concepts

- AsyncTask, Handler — don't execute long-running code on the main UI thread
- Timer, TimerTask
- MediaPlayer, WebView, GPS
- SharedPreferences, PreferenceManager
- LocationManager
- Need to request permissions for certain things in AndroidManifest.xml
- Testing best practices (todo)
- Native code (JNI)?
- Nine-patch images: a stretchable bitmap image; Android automatically resizes to accommodate the view size
- Themes: Holo Light, Holo Dark, Holo Light with dark action bar
- Styles: you can create styles and apply them to widgets in a manner similar to CSS
- Logging (Log.i, Log.e, etc.)
- REST services, internet access

- SQL databases (SQLite)

Android files and folders

- *AndroidManifest.xml*
  - You describe your application in *AndroidManifest.xml*
  - As its name implies, this is an XML configuration file
- Directories in an Android project
  - *src*
  - *res/drawable* - static images
  - *res/layout* - layout files
  - *res/menu* - menu layout files
  - *res/values* - strings.xml

Activity

- an Activity is a controller class (in the MVC sense)
- an Activity generally corresponds to a single Android screen
- need to add each Activity to *AndroidManifest.xml*
- you specify the “launcher” activity for your class in the *AndroidManifest.xml* file (i.e., the main class)
- all other activities are launched with Intents
- fragment hosting - an activity provides a spot in its view hierarchy where the fragment can place its view

The Activity lifecycle

- it's important to know the Android Activity lifecycle, i.e., which methods are available, and when they are called

- in my own experience Android is like Drupal or Sencha in that you implement predefined "callback" methods to do your work
- an activity can be in one of four states (more or less):
    - *Active* - started, and running in the foreground
    - *Paused* - started, is running and visible, but something is overlaying part of the screen
    - *Stopped* - started, running, but hidden by another activity the user is using
    - *Dead* - activity was terminated, such as due to insufficient ram

Working with state changes

(Most of these notes come from the free version of the book, *Busy Coder's Guide to Android Development*.)

- You need to be able to save your application instance state quickly and cheaply
- Activities can be killed off at any time, so you have to save state more often than you might expect
- Think of this process as "establishing a bookmark," so when the user returns the state will be as they left it
- Saving instance state is handled by `onSaveInstanceState(Bundle)`
- The default implementation of `onSaveInstanceState` will (probably) save things like the mutable state of widgets that are being displayed, like the text in a `TextView` (but it won't save whether or not a `Button` is enabled or disabled, or, as i've learned, a background image on a widget)
- You can get that instance state in `onCreate(Bundle)` and `onRestoreInstanceState(Bundle)`
- In some activities you won't have to implement `onSaveInstanceState` at all; this depends on your activity and what data it needs, etc.

The Activity `onCreate` method

- `onCreate` is called when an Activity is created
- OS calls this method "after the Activity instance is created but before it's put on a screen"
- things you can/should do in this method include:
  - inflate widgets
  - put widgets on screen
  - get references to widgets
  - set listeners on widgets
  - connect to external data models
- note: never call `onCreate` yourself
- `onCreate` is called in three situations:
  - when the activity is first started, onCreate is called with a `null` parameter
  - if the activity was running and then killed, onCreate will be invoked with the Bundle you saved with a call to `onSaveInstanceState`
  - when the device orientation changes and you have accounted for that with different layouts

setContentView method

- you will often call `setContentView` in your `onCreate` methods
- `setContentView` inflates a layout and puts it on screen

onDestroy

The `onDestroy` method may be called:

- when the activity is shutting down, because the activity called `finish()`
- `onDestroy` is mostly used for cleanly/properly releasing resources you created in `onCreate`

- because Android shut it down (such as when needing ram)
- note: `onDestroy` may not get called if the need for ram is urgent.

onStart, onRestart, onStop

- `onStart` is called (a) when an activity is first launched, or (b) when it's brought back to the foreground after having been hidden
- `onRestart` is called when the activity is stopped and is now restarting (just after `onStart`)
- `onStop` is called when the activity is about to be stopped

onPause and onResume

`onPause`:

- `onPause` is called when the user is taken away from your activity, such as the starting of another activity
- if you have resources locked up, release them here (background threads, camera, etc.).

`onResume`:

- `onResume` is called just before your activity comes to the foreground, either after:
  - initial launch
  - being restarted from a stopped state
  - after a pop-up dialog was shown
- `onResume` is a good place to refresh the UI, such as when polling a service, or if a pop-up dialog affects the view, etc.

Bundle

- a `Bundle` is passed into the `onCreate` method

- as you'll see, it's also passed into other Android lifecycle methods
- an Android Bundle is a map/dictionary data structure that maps keys to values (i.e., key/value pairs)
- a Bundle can contain the saved state of your views (among other things)
- you can save additional data to a bundle and then read it back later
- has methods like `putInt`, `putSerializable`, `getInt`, etc.

Fragment class

- like an `Activity`, a `Fragment` is a controller class
- fragments were introduced in Android 3.0 when they began to support tablets
- tablets required more complicated/flexible layouts, and fragments were the solution
- fragments let you create small widgets that you can plug into larger views
- said another way, fragments help separate the ui into building blocks
- usually a fragment manages a ui, or part of a ui
- an activity's view contains a place where a fragment will be inserted
    - an activity is said to "host" a fragment by providing a spot in its view where the fragment can place its view
    - an activity may have several places for fragments
- an activity can replace one fragment with another fragment
- the Big Nerd book offers this advice: always use fragments (AUF)
- a Fragment can use `getActivity()` to get a reference to its Activity
- fragments are managed by the FragmentManager of the hosting Activity
- FragmentManager - responsible for calling the lifecycle methods of the fragments in its list
- to use fragments, your Activity must subclass FragmentActivity; AppCompatActivity is a subclass of FragmentActivity

Layouts (Containers)

- you can create your UI views using XML or Java code, but XML is the preferred approach
- of course XML layouts are verbose, but a nice thing is that they work well with the Android Studio designer
- Android Studio also gives you helpful hints when you're searching for attributes to control your views (so it's not like you have to memorize every possible attribute)
- widgets in your layouts are managed by either an Activity or a Fragment
- Android has the following types of layouts (there may be a few more; I've used these so far):
    - ConstraintLayout
    - LinearLayout
    - RelativeLayout
    - FrameLayout
    - RecyclerView
    - ListView
    - GridView

LinearLayout

- in a LinearLayout, widgets and child containers are lined up in either a column or a row, like a FlowLayout in Swing
- a LinearLayout has five main controls:
    - orientation
    - fill model
    - weight
    - gravity
    - padding

RelativeLayout

- a RelativeLayout lays out widgets based on their relationship to other widgets in the container
- RelativeLayout has many configuration options that let you position widgets relative to each other, including these boolean values:
  - `android:layout_alignParentTop` - the widget's top should align with the top of the container
  - `android:layout_alignParentBottom` - the widget's bottom should align with the bottom of the container
  - `android:layout_alignParentLeft` - the widget's left side should align with the left side of the container
  - `android:layout_alignParentRight` - the widget's right side should align with the right side of the container
  - `android:layout_centerHorizontal` - the widget should be positioned horizontally at the center of the container
  - `android:layout_centerVertical` - the widget should be positioned vertically at the center of the container
  - `android:layout_centerInParent` - the widget should be positioned both horizontally and vertically at the center of the container
- it also lets you specify a widget's position relative to other widgets:
  - `android:layout_above` - the widget should be placed above the widget referenced in the property
  - `android:layout_below` - the widget should be placed below the widget referenced in the property
  - `android:layout_toLeftOf` - the widget should be placed to the left of the widget referenced in the property
  - `android:layout_toRightOf` - the widget should be placed to the right of the widget referenced in the property
- (there are more attributes than those. those came from an old version of a book titled, "The Busy Coder's Guide to Android Development")

Common attributes in layouts

- `match_parent` - the view will be as big as its parent
- `wrap_content` - the view will be as big as its contents require
- `@+id` - the actual id will be in *gen/R.java*, inside a `public static final class id { ...`
- `gravity`
- more (todo) ...

UI Components/Widgets

- ActionBar -
- Dialogs -
- Toasts - short lived popup messages
- Menus - don't use these any more, use the ActionBar

Standard widgets are:

- Button
- TextView - use for labels (like JLabel)
- EditText - editable text field (don’t forget you can set keyboard/input options)
- Checkbox
- RadioButton, RadioGroup
- ToggleButton
- Spinner ...
- Picker (DatePicker, TimePicker)

Toast

A *Toast* is a short-lived message that appears in a little popup window. Create a Toast like this:

```
Toast.makeText(getActivity(), "Click!", Toast.LENGTH_SHORT).show();
```

- you use Toasts to show messages to users, such as indicating that something was saved.
- i also use Toasts for testing new code, like this:

```
@Override
public void onListItemClick(ListView listView, View view, int position, long id) {
    Crime crime = (Crime)(getListAdapter()).getItem(position);
        Toast.makeText(getActivity(), "Click!", Toast.LENGTH_SHORT).show();
    }
```

- you can set the *gravity* on a Toast:

```
Toast t = Toast.makeText(getActivity(), "Click!", Toast.LENGTH_LONG);
t.setGravity(Gravity.TOP, 0, 0);
t.show(); ...
```

Snackbar messages

Snackbar messages are like Toasts, but they're shown at the bottom of the display and attached to a view. You can create and display a Snackbar message like this:

```
Snackbar.make(
    view,
    "going to: " + url,
    Snackbar.LENGTH_LONG
)
.show();
```

Toolbar and ActionBar

- the *ActionBar* was introduced in Android 3.0

- it lets you put button/icon controls on your views. a typical button on a ListView is an "add" button, to let you add a new item
- the ActionBar is still supported, but i think it's being replaced by a Toolbar
- you used to have to use an ActionBarActivity to use an ActionBar, but you don't have to do that any more (as of Version ? (todo))
- the Toolbar is newer than the ActionBar, and gives you more control than the ActionBar

Intents

- you use an *Intent* to launch other activities
- here's a simple example:

```
Intent i = new Intent(getActivity(), ImagePagerActivity.class);
startActivity(i);
```

- here's another example where i pass an "extra" when starting a new Activity:

```
Intent i = new Intent(getActivity(), ImagePagerActivity.class);
i.putExtra("POSITION", position);
startActivityForResult(i, 0);
```

Android support library

From the Support Library docs:

- When developing apps to support multiple API versions, Support Library provides a way to have newer features on earlier versions of Android, or gracefully fall back to equivalent functionality
- Leverage these libraries to provide that compatibility layer

- The Support Libraries also provide additional convenience classes and features not available in the standard Framework API for easier development and support across more devices
- Originally a single binary library, the Support Library has evolved into a suite of libraries

Furthermore, “Here are the guidelines for when to use support library classes in place of Framework APIs”:

- If you want to support a recent platform feature on devices that are running earlier versions of Android, use the equivalent classes from the support library
- More sophisticated support library classes may depend on one or more additional support library classes, so you should use support library classes for those dependencies (use `ViewPager` with `FragmentPagerAdapter` or `FragmentStatePagerAdapter`)
- If you do not have a specific platform feature you intend to use with your app in a backward compatible way, it is still a good idea to use support library classes in your app (ex: use `ActivityCompat` instead of the framework `Activity` class, so you can take advantage of newer features later on)

Android command line

I’m pretty weak on the Android command line right now, so I’ll just list a few of the commands I have used:

```
adb logcat
adb shell
adb push image1.jpg /data/data/com.alvinalexander.myapp/files
```

I see Android Studio run some of the following commands. It uses a command like this to install a new version of my app onto the emulator or physical device I use for testing:

```
pm install -r "/data/local/tmp/com.bignerdranch.android.criminalintent09"
```

Different ways to run Java threads

Here are a few ways to run Java threads in Android. First, the Java 8 lambda syntax using a `Runnable`:

```
Runnable runnable = () -> {
    // your code here ...
};
Thread t = new Thread(runnable);
t.start();
```

Or the Java 8 `Thread` lambda syntax (without a `Runnable`):

```
Thread t = new Thread(() -> {
    // your code here ...
});
```

You can also use this lambda approach if you don't want/need a reference to your thread:

```
new Thread(() -> // your code here).start();
```

If you can't use Java 8 lambdas — or don't want to — here's the old thread syntax using a `Runnable`:

```
// pre java 8 lambdas
new Thread(new Runnable() {
    public void run() {
        // your code here ...
    }
}).start();
```

Here's the old syntax without using a `Runnable`:

```
Thread thread = new Thread() {
    public void run() {
        // your code here
    }
}
thread.start();
```

You can also create a class to extend a `Thread` and then run it, like this:

```
public class MyThread extends Thread {
    public void run() {
        // your code here
    }
}
MyThread myThread = new MyThread();
myTread.start();
```

Here’s an approach that uses an `AsyncTask` with a `Runnable`, from the link shown:

```
https://stackoverflow.com/questions/15472383/how-can-i-run-code-on-a-background-thread-on-android
AsyncTask.execute(new Runnable() {
   @Override
   public void run() {
      //TODO your background code
   }
});
```

Code snippets

This code shows how to determine which item in a `ListView` was selected:

```
@Override
public void onListItemClick(ListView listView, View view, int position, long id) {
    Crime crime = (Crime)(getListAdapter()).getItem(position);
        Toast.makeText(getActivity(), "Click!", Toast.LENGTH_SHORT).show();
    }
```

As shown, this code shows one way to show a Toast message:

```
Toast.makeText(getActivity(), "Click!", Toast.LENGTH_SHORT).show();
```

That’s all for now

Reporting live from Boulder, Colorado, that’s all for now, but I’ll continue to add more Android tips as I learn them.