Kevin Chen

1/2/17

Android Development Notes

Introduction

* What you need to know before starting this course
  + Android is a software stack, which includes the operating system, the application framework. Program with Java and XML
  + Java coding standards are based on Java 6 and 7. Advanced architectures.
  + Programming with XML: App resource files.
  + This course uses Android Studio.

Android Development Fundamentals

* Exploring the history of Android
  + Android is a complete software platform: includes OS and an application framework.
  + The Android SDK: Compiler, debugging tools.
  + Version 2.2: Froyo is the oldest version still used commonly today.
  + Version 3.0, Honeycomb, was optimized for tablets.
  + Version 4.0, Ice Cream Sandwitch, unified the tablet and smartphone SDKs.
  + Version 6.0, Marshmallow, is the latest version of Android.
* Exploring the Android app framework
  + Android software stack: Android is based on the Linux Kernel. On top of the kernel is the Android runtime and libraries that enable the operating system’s behaviors. The application framework is next. At the top are the apps.
  + Linux kernel: Linux with a set of drivers designed to interface with some part of the android device. The OEMs (the manufacturers of the devices) customize these drivers and make them work for their android devices.
  + Android runtime: includes core libraries and the virtual machine (originally Dalvik VM, but afterwards ART VM). ART uses ahead of time compilation instead of the just in time model used in Dalvik, which means that apps are compiled into machine code upon installation rather than waiting until features are used for the first time, so apps run a bit faster.
  + Libraries: You include libraries to manage all sorts of features on Android.
  + Application framework: modules (e.g. activities and content providers) for controlling all the different components of your app.
  + Applications: every Android device has a minimum set of apps. Devices licensed to use Google Services will have various Google apps installed as well.
  + Developers put together apps using components. Each component is implemented in Android as a Java class.
  + Component types: activities represent UI screens, views manage display and user interactivity, services perform background jobs (they are invisible), broadcast receivers react to system messages, content providers provide managed access to data.
* Programming with Java
  + The Android compiler transforms your Java code into bytecode for the Android Runtime.
  + The Android SDK supports a custom Java implementation.
  + The Android SDK implements most standard Java packages.
  + Android’s custom APIs: Android-specific features are in custom packages. These begin with android. Example: android.widget. Look at API reference for complete listing.
  + Android built process: Source code (.java) and Library code (.class) are compiled by the standard Java compiler, Java C. This result in Java bytecode (.class). Optionally, the bytecode can be processed and minimized by a tool named ProGuard. Next step is to turn Java bytecode into DEX bytecode, which is a format that’s optimized for Android (can be executed by Darvik runtime and ART runtime).
  + DEX files are what you distribute in your application package. The device’s runtime reads these files and translates them to machine code.
* Install Android Studio on Windows: Install Java SDK, Java Runtime (JavaRE) and Android Studio
* Configure the Android SDK
  + Click on Configure, then SDK Manager. Then select the checkbox for “show package details”, which shows the components that have been installed.
  + Under Android 6.0, select Google APIs, Googel APIs ARM EABI v7a System Image, and Google APIs Intel x86 Atom\_64 System Image.
  + Then go to the SDK Tools tab, and select Intel x86 Emulator Accelerator (HAXM installer) and Google USB Driver, the latter if on Windows.
  + Then click Apply and confirm.

Getting Started

* Create your first Android app
  + Go to the Android Studio welcome screen, and click Start a New Android Studio Project. Specify the Application name (e.g. Hello World) and Company Domain (e.g. hsport.example.com. Any app that belongs to this domain can’t be submitted to the Google Play Store). This generates a package name, which is also known as the app identifier: provides a globally unique identity for the app, so only one app for each identity can be hosted on the Google Play Store. Can be customized by clicking on “edit”. Select project location.
  + Select the minimum SDK that you want your app to support. “Help me choose” can help, but the info can be out of date there (instead go to developer.android.com/about/dashboards/index.html).
  + Select the type of activity you want. Basic Activity adds a Floating Action Button and Java Code when the user presses it. Empty Activity only gives you a bit of text.
  + Enter Activity Name (becomes the name of a Java class), Layout Name (for XML file that describes the appearance of this screen), Title, Menu Resource Name (XML that describes an option menu). Can create the screen by using a fragment (can easily adapt your app to tablets or other sized devices). Click finish.
  + Progress at bottom shows that content is being indexed. If you see errors after waiting for a while, go to Build -> Clean Project.
* Create virtual devices for testing
  + Select Android -> AVD Manager.
  + Then select Create Virtual Device. Choose a device definition. Then select the version of Android (API level) that you want to emulate.
  + Then select the AVD Name, the type of device, and API Level. Then select startup size and orientation (default are good). Enable Device Frame to make it look cosmetically better. More options under advanced settings. You should enable keyboard input to type values from your computer’s keyboard.
  + Then select it and click the run button to run the device. In the output, look for the message “HAXM is working and emulator runs in fast virt mode” which says HAXM is making the virtual device run as fast as it can.
  + Now your virtual device is ready to run an app. Click “run” on your project. The dialog will show that it is waiting for the ADB service, which will connect Android Studio to the device. Select your device.
* Connect a physical device for testing
  + Default USB mode is charging. Change it to photo transfer.
  + On MSFT Windows, you need to have a driver installed to connect to a physical device (this might be done automatically). Check if it is connected correctly by going to Control Panel -> Device Manager -> Portable Devices. Check to see if your device appears there.
  + You may need to update the driver software. First download it from the vendor. (For Nexus, this can be done through the SDK Manager: Android Studio welcome screen -> configure -> SDK Manager -> SDK tools -> Google USB Driver.) Then right click the device in the Device Manager, and select Update Driver Software. Then select Browse my computer for driver software, and select the path: C:/Users/*Name*/Android/sdk/extras/google/usb\_driver.
  + Turn on USB debugging on the device. Select Settings -> About phone -> Build Number x 7. Then go back a screen and select Developer options. Turn on USB debugging (at the bottom of the list) and stay awake (for convenience).
  + When connected, you must give USB debugging authorization on the device. (To get rid of current authorizations, select revoke any existing authorizations in the Developer Options.)
  + Verifying the device: First way to verify is to open the Command Prompt. If you have access to the command line tools, you should see output when typing in adb (see below lesson for doing so). If you don’t, then change the directory to the platform-tools folder in the sdk folder. If the device is correctly attached, then adb devices should return a unique identifying string. Second way to verify is to go to Android Studio, then select Android Monitor, then see your device listed correctly in the top left drop down menu.
* Add command line tools to the path
  + In Android\sdk\platform-tools, adb.exe is the Android debug bridge, a software that lets you communicate with a physical or virtual device. fastboot.exe is used when configuring an Android device.
  + In Android\sdk\tools, there is emulator.exe.
  + The above commands are very important. You want to be able to use them from any directory. Do so by going to Control Panel -> Environment Variables. In the user variables, create a new (or edit the existing) variable named PATH, and add the file paths to the above two folders.
* Explore the SDK’s command line tools
  + In the command prompt, the command “android” opens the Android SDK Manager Application. Manage components you have installed.
  + android avd opens the Eclipse version of the AVD manager to manage your virtual devices.
  + android list avd displays a list of all your virtual devices.
  + emulator –avd *DeviceName* launches the virtual device. (The command doesn’t finish in the command window until the virtual device is closed.)
  + adb reboot restarts the device. (This doesn’t work well with a virtual device.)
  + monitor opens the Android Device Monitor. This app has a Devices view from which you can choose the device you want to work with. Various tabs to manage the device.
  + android create project --target *n* --name *AppName* --path *Path* --activity *ActivityName* --package *PackageName* creates a new Android project. (Target *n* indicates the *nth* SDK image. *Path* is where the project will be created; use forward slashes. *ActivityName* is the main activity’s name. *PackageName* is the unique app identifier).
* Use the developer documentation
  + developer.android.com/develop contains the documentation in the reference tab.
  + To download the documentation, go the SDK manager. Then go to SDK tools, and download. “Documentation for Android SDK”. To access the documentation, go to the sdk folder, and click on docs. Then click on the index page.
  + Press Ctrl + Q on a class or method in a project’s Java files to open a simplified version of the documentation.
  + Press Ctrl and move your mouse over a class name to find more info about a class.

Working with Android Studio Projects

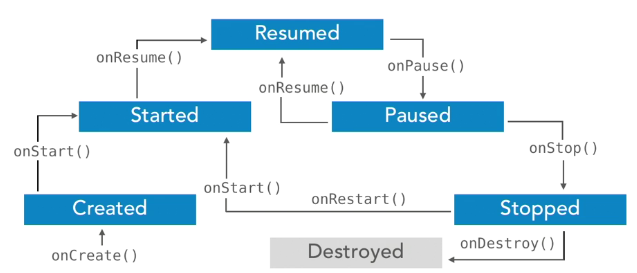
* Explore the project structure
  + In the Project Window within Android Studio, by default, the filtered view named Android is selected. But this view doesn’t reflect the actual underlining structure on the hard disk. Switch to the Project view (using the drop down menu) to see the actual underlining structure.
  + app folder represents a module. Contains a src folder, which contains a folder named main. This contains a Java folder, resource folder, and the AndroidManifest.xml file (an XML file that you use to configure your app). The Java folder contains folders for your packages. The package is reflected both in the java activity file and in AndroidManifest.xml. The package is also registered in build.gradle.
  + The res folder contains all your resource folders. drawable contains .png and .jpg files to represent graphics in your app. layout has XML files that describe what your screens look like (initially, one is called activity\_main and the other content\_main. activity\_main servers as the container for the whole screen description, and content\_main contains UI elements). The menu subfolder contains XML files that describe menus. mip-map folders are populated with the launcher graphic, which is displayed on the device’s launcher screen to show the user where to launch the application. These folders are labelled with dpi labels (which represents pixel density). During runtime, the app framework selects the appropriate version (based on device’s dpi) to display. values subfolder contains XML files. One file within the values subfolder is string.xml, which describes screens and gives them identities, which makes it possible to localize or change your app for different languages. values21 folder has files that override settings when running on a device with API21 or later. valuesw820dp override settings when device has larger screen (at least 820 device independent pixels on one side).
  + The Android view makes it faster and easier to do Android development.
* Manage Gradle build scripts
  + When you create an Android Studio project, it is populated with two Gradle scripts and a couple of other configuration files.
  + Open the Gradle Scripts folder, and go to local.properties. This contains the location of the Android SDK. This value can be modified directly or done through a UI by going to File -> Project Structure -> SDK Location.
  + gradle.properties is completely commented out. This manages memory for Gradle during the build process.
  + The two gradle scripts are both called build.gradle. One is for the project, the other is for the module (or the app itself). The project one has some important settings for the Gradle build process.
  + What is a module: Go to Project Structure -> app (in Module category). This represents your application. Changes here will be made in the build.gradle file.
  + The build.gradle app for the module contains various settings for setting your Android versions and code dependencies. buildTypes is a feature of the build process where you can generate multiple versions of an app for distribution. The dependencies block is a listing of all the directories and jar files you need to build your application. But Android doesn’t use actual jar files; instead it uses Maven dependency system, which is a system of repositories that store code for use in Android. compile fileTree(dir: 'libs', include: ['\*.jar']) says to include any jar files that are included in a libs folder under Java. The Appcompat library refers to a support repository, which contains Java classes and libraries that can be used instead of versions in the core SDK so that you can write apps that work on older versions of Android. The design library contains definitions of new tools that are available for Marshmallow.
* Manage the app manifest file
  + The AndroidManifest file is an XML file where you define your activities and control their behaviors and appearance, and also where you can grant permissions that allow your application to execute critical functions.
  + Root element is named manifest. Contains a child element named application and can contain other child elements. Begins with a prefix of android. Package attribute points to app’s default package (aka application ID).
  + Application element controls aspects of your element: allowBackup to back up app data on Google, icon controls graphical element displayed on device’s launcher screen, label is the string the user sees as the app’s name, supportsRtl configures whether you want to support right to left text rendering, theme controls the application’s appearance.
  + Within Application element is activity element, which represents a screen. Name of the activity represents a Java class that determines that screen’s behavior. label and theme overrides the settings for this activity. An intent-filter determines when the activity is brought to the screen. An activity with an intent-filter that has an action of MAIN and a category of LAUNCHER is the app’s launcher activity (or the activity that you see first).
  + Configuring permissions: these are child elements of the root. Add a uses-permission element for each permission you want to add.
  + (Reformat code by pressing ctrl + alt + l or Menu -> Code -> Reformat Code.)
  + Some attributes may use resource IDs (these begin with a @ symbol), which are shown when you click on that attribute.
* Use dependencies and support libraries
  + Dependencies based on the Maven system. A Maven repository is a library of libraries. Examples: Appcompat gives you features that you can add to an app that would work all the way back up to API level 7 devices. Design library includes components for visual displays.
  + To use these features, you must install the Android Support Repository, which is done through the Android SDK Manager (in the SDK Tools section).
  + During compilation, the build process looks at the directives, looking for unique identifies for the libraries and versions you want to use. It looks at an index that’s included with the SDK installation. If those libraries are found, it gets them from the repository. If it doesn’t, you can get libraries from a remote repository.
  + To include a dependency for a library from a remote repository: File -> Project Structure -> app -> Dependencies tab -> Plus icon ->1. Library Dependency. Then choose your dependency, and hit okay.
  + Where the library files come: open build.gradle file for the project, and you see a repositories list, which is a list of Maven repositories that the build process can use to go get requested libraries. It includes the jcenter() Java method, which returns an instance of a class named MavenArtifactRepository, which is a reference to a website that contains the libraries.

Define the User Interface

* Create layout resource files
  + The launcher activity is the activity that is loaded when the user launches the app.
  + As an activity is created, the method onCreate is called. One of the most important tasks is to choose the activity’s layout, which is done through the method setContentView(), passing in a resource ID for a layout file.
  + There is also a secondary (nested) layout, which is shown in the include element. Points to content\_main.xml.
  + When creating new layouts, it is recommended to first make a copy of a layout file that is already working.
* Modify material design themes and styles
  + MaterialDesign is a comprehensive guide for visual, motion, and interaction design. Used for both Android and web apps.
  + The theme of your application (as specified in the manifest file) can inherit from a material design. You can take a material design and modify some of its values. Look online to explore the various material designs.
* Lay out screens with ViewGroup and View
  + Each element in the XML file represents a Java class.
  + All visual widgets are described in Java code as subclasses of the View class. Example of a subclass is TextView. The XML is interpreted at runtime as a request to create an instance of the Java class and add it to the activity.
  + A ViewGroup is a subclass of View. ViewGroups are layout contains. Examples (subclasses of ViewGroup) include LinearLayout (lay out horizontally or vertically), RelativeLayout (relative to container or other objects), GridView (2D scrollable grid), ListView (vertical list of scrollable items). AbsoluteLayout, DrawerLayout, ViewPage. From the support design library: CoordinatorLayout AppBarLayout.
* Units of measurement
  + Set dimension values for visual components. Primarily set in XML-layout files or programmatically. Most can be changed during runtime with Java code.
  + Device-Independent Pixels (dp): control position and size of contains and widgets so that their appearance would be the same regardless of the size and density of the actual device. On a 160 dpi screen, each dp is mapped to a physical pixel.
  + Scale-independent pixels (sp): similar to dp, but used to specify font sizes.
  + Dimension resource files define all of your dimensions. A dimension resource is defined in an XML file placed under the values folder under resources. (dimens.xml).
  + Setting fixed dimensions with Java (good for runtime adjustments): modify the layoutParams fields in each view object. *viewname*.getLayoutParams().height = 100 to modify height. Can modify width and other properties as well. But this is using absolute (physical) pixels. To use dp, convert the pixels to dp with TypidValue.applyDimension. One of the arguments requires screen density and resolution, which can be obtained from getResources().getDisplayMetrics().
* Using LinearLayout containers
  + Lays out child objects horizontally or vertically. Activity\_main is the parent layout and content\_main defines the objects that appear in the main content area.
  + In content\_main, add an android:orientation attribute and sets it value to horizontal or vertical.
  + The gravity attribute applies a setting to the objects within a container. Set it to center\_horizontal if you want to center the objects.
  + (In the design view of the xml file, the icon named Orientation in Editor allows you to switch between Portrait to Landscape.)
  + (Select some text and press Ctrl + D to duplicate that code.)
  + (The android:id attribute of each view must be unique.)
  + You can have a LinearLayout within a LinearLayout. Whenever adding items in the design view, verify they are added in the right place by looking at the XML and seeing which element the item is a child of.
  + android:layout\_margin=“*somepixels*” adds a margin of *somepixels* in all four directions around a view to help separate it a bit from other views.
* Using RelativeLayout containers
  + Lets you anchor objects to a container’s edges or to other objects within the container. Also uses activity\_main.xml and content\_main.xml.
  + (Quick way to convert a literal string to a string resource: Select the literal string and press alt + enter. This opens an Intention Action menu. Choose extract string resource. Give it a resource name.)
  + Anchor an object to the a container’s edge by making it a child of the container, and adding the attribute: android:layout\_alignParentTop=“true” (to align it to the top). Set android:layout\_alignParentRight and android:layout\_alignParentEnd true for right. Left and Start for left. Bottom for bottom.
  + Anchor objects vertically to each other by designating the tallest component to serve as the anchor. Add the following additional attribute to the other objects: android:layout\_alignBaseline=“*IDofanchor*”.
  + You can anchor an object to other objects lets you to expand that object so that it fills up the available space. Add attributes to the object: android:layout\_toRightOf=“*IDofRightObject*” and android:layout\_toEndOf=“*IDofRightObject*”. Set toLeftOf and toStartOf for left object. Set below or above for below object or above object.
* Add views to a view group with Java.
  + Allows you to add views dynamically.
  + First need to add an id to your layout. Add the attribute: android:id=“@+id/*name*”. When you do this, there’s a background process in Android Studio that generates a Java class with a unique numeric value, for each ID. (This is how you get the reference.) The location of this file can be found by searching everywhere (Hit left shift button twice) R.java, and selecting the file your package folder – OR – by going to *Application*/app/build/generated/source/r/debug/*package*/R.java.
  + Go to your activity and get the ViewGroup: *ViewType* *name* = (*ViewType*) findViewById(R.id.*idname*). Example: LinearLayout layout = (LinearLayout) findViewById(R.id.content\_layout);
  + Create LayoutParams for each view you want to create. ViewGroup.LayoutParams *name* = new ViewGroup.LayoutParams(*width*, *height*). (This is just one of the few constructors.) Set *width* and *height* to ViewGroup.LayoutParams.WRAP\_CONTENT if you want it to take up just enough space to fit its own internal content.
  + Then create the views: *View* *name* = new *View*(*context*). Note that an activity is a context, so you can just pass in this. Example: Button button = new Button(this); Then use the setLayoutParams() to set the layout params. Set any additional settings. Then add the view to the layout. Example: layout.addView(button);
  + Anything in the layout is executed first, and then anything in the onCreate method is executed later.
* Display and collect text values
  + Use EditText to collect text values. Give this view an id.
  + When you want to use the EditText’s text value (e.g. inside the event handler for a button), get a reference to the EditText. EditText *name* = (EditText) findViewById(R.id.*value*); Get the value using *name*.getText().toString();
  + Use TextViews to display text. Get its reference, and then call it’s setText() function, passing in the string value.
  + (Get the string value of a resource object by using the method getString(*resourceid*). Example: getString(R.id.name))
  + ScrollView can be used to allow the user to scroll through text (if the device cannot display all the text at once). Wrap the TextViews and/or other views you want to be scrollable within a ScrollView. Set its android:layout\_width and android:layout\_height attributes to match\_parent if you want it to fill up the size of the parent.
* Display messages with Toast and Snackbar
  + Displaying a Toast: Toast.makeText(*context*, *message\_to\_display*, *length*).show(). For context, pass in MainActivity.this. For length, pass in the constants Toast.LENGTH\_LONG or Toast.LENGTH\_SHORT. A Toast message is managed by the application framework, not directly by the application UI, so it can survive even the app being closed down.
  + Displaying a Snackbar: Snackbar.make(*view*, *message\_to\_display*, *length*).setAction(“Action”, null).show(); For the view, pass in the coordinator layout that is managing the whole screen. (You have to give it an id, and reference it in Java. You don’t have access to the layout until the setContentView in the onCreate() method has been called.) A Snackbar message can be dragged away to dismiss it.
  + (Ctrl + / comments out the selected code.)
  + (The onCreateOptionsMenu opens a menu resource file and displays the menu. The onOptionsItemSelected is called when any item is selected from the menu.)

Work with Events

* Explore the activity lifecycle
  + Typically the activities of a single app are all in the same task. When you launch an activity, that activity becomes part of a task. If the activity launches a second activity, the first activity goes to the back of a stack known as the backstack, making the second activity visible. Launching another activity results in creating another task. If the OS has low memory, it can remove certain non-active tasks from memory.
  + Activity stacks rule: only one activity is active at any given time (though there can be more than one visible activity). One activity can be the last activity, or the most recent activity. The last activity becomes active when the current activity is closed or finished. Other activities are queued as previous activities on the backstack.
  + Activity states: Active/resumed: top of the stack, visible, and interactive. Paused: can be visible but without focus. Stopped: back of the stack, not visible. Inactive: completely removed from the activity stack.
  + The activity last lifecycle: Each state is associated with an event, and each event is associated with a method. These methods are defined in the activity class, and can be overridden. When an app is launched, it goes through the created, started, and resumed states. When it is in the resumed state, it is visible and interactive. When the user shuts down the activity, it goes to the Paused, then Stopped, then Destroyed states. Destroyed state is where it is removed from memory. (It is still visible in Paused. It is not visible in Stopped.) The app can also go from Stopped back to Resumed and from Paused back to Resumed.



* Trace lifecycle events with Logcat
  + Trace the events in your code using Logcat. Only use this for developing and testing code. Remove this for distribution.
  + The Log class has a set of methods you can call to send messages to the logcat console. Log.d() is for sending debugging messages. This takes in a tag (you can filter by tag in the console) and the message. One common approach is to tag the message by the current class by either using a literal string or using the method getClass.getSimpleName(); or using a constant.
  + (A fast way to replace a literal string with a constant is by selecting the string and then selecting Refactor -> Extract -> Constant.)
  + To view the logcat console, open the Android Monitor (Ctrl + 6). Filter by selecting the appropriate log level and using the search bar for the tag.
* Handle view events in layout files
  + Create the appropriate attribute in the layout file to handle the event for a view. Set its value to a name that will become a Java method. Example: this attribute is for handling when a button is clicked: android:onClick=“buttonClickHandler”. Press alt + enter -> Create onClick event handler to create the event handler in Java code.
  + The event handler method has a parameter, a view, which is the view that triggered the event.
* Handle View events with Java code
  + Create an id for the view that you want to add an event handler to.
  + (shift + f6 is the keyboard shortcut to rename.)
  + Get a reference to the view using getViewById. Then call the view’s method that sets a listener. For example, for a handling a button press, call button.setOnClickListener(), passing in an instance of an interface named OnClickListener. (Using an anonymous class is the most convenient.)
* Handle changes in screen orientation
  + A configuration event is an example of an event handled automatically by the application framework, but you can choose to handle them yourself with Java code. An example of a configuration event is the change in screen orientation.
  + By default, changes in screen orientation is done by destroying the activity in memory and rebuilding it from scratch.
  + How you handle this event: go the manifest file and go to the activity you are working with. Add an attribute: android:configChanges=“orientation|screenSize” (this attribute is used to work with various configuration changes).
  + Override the method onConfigurationChanged(). You receive a new object, newConfig. You can inspect the orientation of newConfig with the newConfig.orientation attribute. Can check if it is equal to Configuration.ORIENTATION\_PORTRAIT.

Display Images

* Create a custom launcher icon
  + There are no icons that are simple squares; there’s always a certain amount of transparency. The colors tend to be flat; you don’t see a lot of texture, and that’s to fit with the material design visual language.
  + Each Android Project you create, be default, has a launcher icon of Andy the Android. You want to replace this to brand your app.
  + To add an image to your application, select the app module in the project window -> New -> Image Asset. Asset type: launcher icons. Foreground: image. Select the image file. Make sure the resource name is called ic\_launcher There are 5 different versions generated, one for each of the standard pixel density buckets. In addition to the launcher icons, you will get a file called ic\_launcherweb.png, which will be useful when you deploy the app to the Google Play Store.
* Display an image resource with ImageView
  + Import the image as an image resource: Copy the image. Right click on the res folder, and hit paste.
  + Add an ImageView component. Add an attribute android:src=“*resourceID*”, using the resource ID of the image. Adjust the width and height of the image (Modify android:layout\_width and android:layout\_height.) and other attributes (e.g. margins). Make sure you use dps, and save them as dimension resources.
  + (layout\_centerHorizontal=“true” will center the image between the left and right borders.)
* Display image resources at runtime
  + Add an ImageView, like in the previous section, but don’t add a source attribute.
  + Get a reference to the ImageView control using findViewById(). Display the resource by calling the instance method setImageResource(R.drawable.*resourceID*) of the ImageView control.
  + Another way is to get a reference to the ImageView. Then create a string and set it to the resourceID. Get the resource corresponding to the resourceID using the method getResources.getIdentifer(*resourceIDString*, *Category*, *CurrentAppPackage*), where *Category* is “drawable” and *CurrentAppPackage* is getPackageName(). Now set the ImageView to the image resource obtained. This is useful if you are starting with a String value (e.g. from a database) and want to get the Resource. But this process can be slow if you’re scrolling a long list of values. So using assets is another approach (see next section).
* Display images from the assets folder
  + Asset files are placed in the assets folder. Unlike resources, they aren’t compiled into the application in any way and identifiers aren’t assigned. Address an asset file using its real name.
  + Select the app module in the project directory. Then go to: File -> New -> Folder -> Assets Folder. Default location is the main folder of the current module. Click finish. The new Assets folder is created under the main folder.
  + Move the image to the assets folder.
  + (When an app crashes during runtime, i.e. RuntimeException, then go to the Android monitor and filter for AndroidRuntime. Look for “caused by”)
  + Open a stream to the file. (InputStream stream = getAssets().open(*imagename*); You need to do getContext.getAssets()… if you’re not calling this method from within an activity class.) This may throw an IO exception, so surround with try-catch, or add throws clause. Then instantiate a Drawable using the stream. (Drawable d = Drawable.createFromStream(stream, null);) And then call the setImageDrawable() method of the ImageView instance, passing in the Drawable.
  + Recommended to use resources folder for single/few images and assets for many images.

Work with Menus and the Action Bar

* Define the options menu with XML
  + Go to the menu\_main.xml file inside the menu folder. This contains a menu and the items on that menu. Use the basic activity template as a guide to creating a menu.
  + Items in the menu XML file should have the attributes android:id, android:orderInCategory (controls where in the menu the item is placed. Higher values go to the bottom. zero means you don’t care where it is placed), android:title, and app:showAsAction (set this to “never”)
  + In the onOptionsItemSelected method, notice that the method gets the id of the menu item that triggered the event using the method getItemId(). This can then be compared to the resourceID of each menu item to see which item was selected.
  + (Alt + F7 will find usages of the selected text.)
  + You can create additional menus, and control which menus are shown in the onCreateOptionsMenu method (which is called as the activity starts up). In the getMenuInflater().inflate() method, pass in the menu you want to use.
* Add items to a menu at runtime (using Java code)
  + The onCreateOptionsMenu has a menu parameter. Call the menu’s instance method of add(). Pass in a group ID (keep this value consistent across all menu items in the same group), an item ID (pass in any constant), an order value (the same kind of value as in the attributes android:orderInCategory), and a string label (use string resource).
  + Then add event handling code in the onOptionsItemSelected() method. Check if a particular item was selected by seeing if the id of the item selected is equal to the item ID (the second parameter you passed into the add() method above) of the particular item.
* Display menu items in the action bar
  + In the menu\_main.xml file, create an item in the menu, and set its app:showAsAction=“always” to always show the item in the action bar. (Can set it to ifRoom, so it shows if there is enough room)
  + Why you use the app prefix instead of android prefix: The Action Bar is replaced with a Toolbar, which is defined from the support library. Generally, attributes in the core Android SDK libraries have the prefix android, but there are times when it uses the prefix app.
  + Replacing the string for the Action Bar item with a graphic: create a graphic specifically for the Action Bar. One way is to go to File -> New -> Image Asset. Choose Clipart and choose an icon.
  + But sometimes you won’t find something appropriate or specifically designed for use with the Material Design visual language. Go to materialdesignicons.com for more graphics. Then go to File -> New -> Image Asset. Change Asset type to Action Bar and Tab Icons. Brows for image file. Change the resource name. Now you can see all versions of the images under the drawable folder.
  + Add an android:icon attribute to the menu item in the action bar., and set its value to the resourceID of the image.

Manage Navigation

* Create new activities and layouts
  + Create a new activity: File -> New -> Activity (and then choose type of activity). Set the hierarchical parent to be the activity that is shown when the user presses the back button from the activity.
  + Changes from MainActivity: at the end of onCreate, there is an additional method setDisplayHomeAsUpEnabled(true), which provides the user one more way to return to the parent activity (it creates an “up” arrow, though in reality, a left arrow, in the action bar for older versions of Android). Two new layout files will be created: one a parent, one a child.
  + The new activity can be seen in the manifest file. The name has a dot in front of it, meaning that the class is a member of the app’s default package. Within the new activity, there is an element named “meta-data”, which contains a “name” and “value” attribute. The value is set to the MainActivity class. This is enough in recent versions of Android to cause the “up” icon to appear. For older versions, the setDisplayHomeAsUpEnabled() needs to be called (as described above).
* Start activities with explicit intents
  + Use the Intent class to navigate from one activity to another activity within the same application. Create an instance of it. Constructor takes in a context and class. The class is the class property of the activity class you will go to. Example: Intent intent = new Intent(getApplicationContext(), AboutActivity.class);
  + Then call the startActivity() method, passing in the intent object.
* Open other apps with implicit intents
  + Implicit intents allow you to share data with other applications on the same Android device.
  + Navigating to a web page: create an intent. Pass in the values Intent.ACTION\_VIEW (shows you want to view something) and Uri.parse(*WebURL*). Whenever you use an implicit intent, you must resolve the activity (or make sure that there’s an app on the device that’s registered to handle the content you are sending). Call the resolveActivity() of the intent object and pass in getPackageManager(). If the method can find an app that can handle the URI, then it will return an object; otherwise, it will return null. Thus, check if the value returned by this method is not equal to null. If so, start the activity by calling the function startActivity(), and pass in the intent.
  + Navigating to an email: Create an array of email addresses to be sent. Create an intent, passing in Intent.ACTION\_SENDTO for sending to an email address. (ACTION.SEND is for sending attachments; ACTION.SEND\_MULTIPLE is used to send multiple attachments.) Call the instance method setData() for the intent, passing in Uri.parse(“mailto:”), which means that this intent will be handed by an email application. Use the putExtra() instance method to put some key value pairs. For the key Intent.EXTRA\_EMAIL, put the email addresses array. For Intent.EXTRA\_SUBJECT, put a default subject. For Intent.EXTRA\_TEXT, put a default email message body. Then resolve the activity (look above for details), and if resolved, then call startActivity().

Support Multiple Screens

* Android market fragmentation
  + Devices have all types of operating system versions because there are many Android device manufacturers.
  + Android 4 is considered to be the start of “modern Android”. Many developers only support Android 4 or above.
* Create alternative resources
  + Sometimes you want the app to look differently in landscape orientation. In the design view, select the layout variants icon in the toolbar, and select create landscape variation. This creates a new XML file in a new folder named layout-land within the res folder. When the device is flipped horizontally, the new landscape XML layout file is used.
  + This is one example of how to use resource directory names to control behavior at runtime. Look at documentation to control behavior under other conditions.
* Define alternative dimension resources
  + Use specially named dimensions resource folders to manage what happens on large-screen devices.
  + By default, there are two dimens.xml files. One of them is the w820dp folder. This file overrides some settings of the default dimens.xml file for devices with at least 820 pixels on one side. You can add some addition settings to override in this xml file.
  + (Ctrl + F11 changes the orientation of the emulator.)
* Create multiple versions of an image
  + Android Studio can only create different versions of icons, but not for images.
  + You need to use another tool to create different versions of an image. One of these tools is the Final Android Resizer, which can be found on Github.
  + Copy all versions of the drawable folder. Then go to Android Studio, and right click on the resources (res) folder in Android Studio, and hit paste.
  + You might get some thumbs.db files. Those are artifacts of Windows; they aren’t part of the Android app, so you can delete them.
* Lay out activities with fragments
  + When creating a new project, select the option to use a fragment.
  + When you do this, there are 3 xml files instead of 2 created. content\_main.xml points to a java class named MainActivityFragment, which has an onCreateView() method that loads its own layout, fragment\_main.xml. This puts together an application layout dynamically at run time.
  + This creates an app with multiple fragments. The main screen is a menu of the various activities in the app. The app is going to look the same on all devices.
  + The Master Detail Flow project is similar to the above app, but it has a two-pane mode for longer devices (tablets), where it displays the menu and the details on the same screen. There are four xml files. When you open the application, ItemListActivity starts up. It looks for a particular element; if it finds it, then it’s in two pane mode, otherwise single pane mode. Two xml files, one for each mode. The one designed for two pane mode has a reference to a fragment, which references an ID of item\_list

(Bottom line: use specially named resource folders to store different versions of a file.)

Work with Data

* Display an array of data in a list
  + string-array is a type of resource. Consists of various elements named items.
  + Can use a list view to display a list of items.
  + Get a reference to all the resources of a project in Java by using the method getResources(), which returns a Resources object. Then can use an instance method of the Resources object to get a particular resource, passing in the resourceID.
  + Next create, an ArrayAdapter, which binds the data to the list. Declare it of the type of the array data. For the constructor, pass in the context (this), the resource ID for a layout file containing a layout to use when instantiating views for each row entry of the list, the ID of the TextView that will display an array element for a single row, and the data it is using (the array).
  + Then get a reference to the list view, and set its adapter to the one created in the previous step.
* Customize list item displays
  + Create a custom xml file for the layout of each row entry of the list. (We’ll call this *list\_item*.)
  + Create a custom adapter class that extends the ArrayAdapter class. The constructor should take in the context, the resourceID of the layout file (the layout file created in the previous step will be passed in), and the list of objects to display.
  + The above class should override the getView(int position, View convertView, ViewGroup parent) method, which is called when the user scrolls up and down in the list. The goal is to take data from the current list entry and display it in one of the visual controls in the layout. To do so, first check if the current view is null. If so, instantiate it using a LayoutInflater: convertView = LayoutInflater.from(*currentcontext*).inflate(*layoutfile*, parent, false); (*currentcontext* can be obtained through getContext(), the *layoutfile* is *list\_item* in this case, parent is the view group, and false is the value for attach as root.) Then take the views of the convertView and set them to the appropriate values as obtained through the list of objects stored in the previous step. Then return the convertView.
  + In the activity Java class, create an instance of the custom adapter, passing in this, the list\_item layout file resourceID, and then the list of objects. Then display in the listview by obtaining a reference to it and setting its adapter.
* Customize text and image displays
  + (How to get a bitmap from an assets folder: open a stream to the assets folder as you would for images…refer to above section “Display images from the assets folder”. The method BitmapFactor.decodeStream(*stream*) will return the bitmap. For the ImageView, use the instance method setImageBitmap(), passing in the bitmap.)
* Create and open a detail activity
  + The user expects to see more details after clicking on a particular item in the list view.
  + Create an activity that the user will see after clicking on a particular item. Set the hierarchical parent to MainActivity, which means that you’ll return to MainActivity when you are done with this activity.
  + In MainActivity.java, create an event handler for when the user clicks a particular item on the list view. This can be done by using the instance method setOnItemClickListener(). Pass in an anonymous class. In the onItemClick() method of anonymous class, create an intent between MainActivity.this and *otheractivity*.class, and then start the activity.
  + Now you need to customize the new activity to reflect what the user has selected (see next section).
* Pass data to a detail activity
  + Use the position parameter in onItemClick to see the index of the item that the user has selected in the list view.
  + Add extras using the instance method putExtra() of the Intent class in order to pass key-value pairs to the new activity. (Good idea to make the key a string label of the type of data you’re passing in.) Use this information in the new activity to display something customized to the item that the user selected. (In the new activity, use an instance method of the Intent object returned by getIntent() in order to access this key-value pair.)
* Return data to a parent activity
  + Return data to the parent activity by using intent extras. You’ll need a constant for each extra that’s access to both the sender and receiver that’s used as the extra name (or the key).
  + In the MainActivity, create a unique private constant integer, which will be used to coordinate the two activities. Currently you are using the startActivity() method to launch the second activity (the activity with the details). Replace that with the startActivityForResult, passing in both the intent object and the new integer. Also create a public String constant that will label the extra being returned from the second activity.
  + In the second activity, add some way for the user to go back (e.g. FAB). In the code that handles the event (e.g. button pressed), create an intent using the default constructor. Put an extra in the intent, using the public String constant as the key and passing in whatever value you want to return to the parent activity. Then call the method setResult, passing in RESULT\_OK (note there is also RESULT\_CANCELLED) and the intent object. Then call the method finish(), which closes the current activity and returns to the parent activity.
  + Back in the main activity, override the method onActivityResult(int requestCode, int resultCode, Intent data); First check that you are coming from the correct second activity by checking if requestCode is equal to the unique private constant integer. Also check that the resultCode is equal to RESULT\_OK. Then use the Intent object however you want (extract the extras to get the data).

Conclusion

* Next steps (Course suggestions):
  + Working with data: “Android SDK: Local Data Storage” and “Building a Note-Taking App for Android”
  + Integrating Google Play Services: “Adding Google Maps to Android Apps”
  + Using web services: “Connecting Android Apps to RESTful Web Services”
  + Supporting Multiple Screens: “Building Adaptive Android Apps with Fragments”
  + Deploying your apps: “Distributing Android Apps”