ATLS 4120/5120: Mobile Application Development
Week 9: Android Development Basics

Now that we've seen the basic structure of an Android app, let's look at how we can create one that does something.

All Android documentation is on their developer web site https://developer.android.com Develop | Reference https://developer.android.com/reference/packages.html

Views
A view is the building block for all user interface components

- The View class is the base class for all widgets **android.view.View** https://developer.android.com/reference/android/view/View.html
  - TextView
  - EditView
  - Button
  - ImageView
  - Check box
  - and many others
- Views that can contain other views are subclassed from the Android ViewGroup class **android.view.ViewGroup** which is a subclass of the View class. Single parent view with multiple children. (ex: RadioGroup is the parent with multiple RadioButtons)
  - Menus
  - Lists
  - Radio group
  - Web views
  - Spinner
  - Layouts
  - and many others
- Access views in your code using **findViewById(id)**
- Android includes many common UI events that you can set up a listener for and assign a handler to. We'll look at them more closely next week.

Views are saved as XML -- eXtensible Markup Language

- XML is a markup language designed to structure data
- XML rules
  - First line - XML declaration
  - XML documents must have a root element
  - Attribute values must be in quotes
  - All elements must have a closing tag
  - Tags are case sensitive
  - Elements must be properly nested
  - XML must be well formed

Widgets
The Android SDK comes with many widgets to build your user interface. We're going to look at 4 today and 5 more next week. https://developer.android.com/guide/topics/ui/controls.html

TextView

- Text views are used to display text **<TextView …/> android.widget.TextView** https://developer.android.com/reference/android/widget/TextView.html
- The **android:textSize** attribute controls the sixe of the text
  - Use the sp unit for scale-independent pixels
  - Scales based on the user's font size setting
- The **setText(text)** method changes the string in the text view
- Not editable by the user

EditText
- Edit text is like a text view but editable **<EditText …/> android.widget.EditText** https://developer.android.com/reference/android/widget/EditText.html
- The **android:hint** attribute gives a hint to the user as to how to fill it in
- The **android:inputType** attribute defines what type of data you're expecting
  - Number, phone, textPassword, and others
  - Android will show the relevant keyboard
  - Can chain multiple input types with "|"
- **getText().toString()** retrieves the String

Button
- Buttons usually make your app do something when clicked **<Button …/> android.widget.Button** https://developer.android.com/reference/android/widget/Button.html
- The **click** event is fired when the button is clicked. Set up the listener and handler to respond to the event.

ImageView
- ImageViews display an image **android.widget.ImageView** https://developer.android.com/reference/android/widget/ImageView.html
- Images are added to the res/drawable folder in your project
- You can create folders to hold images for different screen size densities
- The **android:src** attribute specifies what image you want to display
  - @drawable/imagename
- The **setImageResource(resId)** method sets the image source

Resources

A resource is a part of your app that is not code – images, icons, audio , etc
- You should use resources for strings instead of hard coding their values **android:text="@string/heading"**
  - Easier to make changes
  - Localization
  - @string indicates it's a string in the strings.xml resource file
  - heading is the name of the string
- Use ids for resources you want to access in your code **android:id="@+id/message"**
  - When you add IDs to resources they will have a + sign because you are creating the ID
  - You don't use the + when you are referencing the resources.
  - **findViewById(id)** uses the id to access the resource

Layouts

Layouts define how you want your view laid out. Constraint layouts were introduced in AS 2.2 in the fall of 2016 while still in beta. As of 2.3.3 they are the default layout when you create a new project so we will be using them. For now we'll leave AutoConnect on so it will handle our constraints for us.

Launcher Icons

- Launcher(app) icons represent your app. They appear on the home screen, launcher window, and can also be used to represent shortcuts into your app
- https://developer.android.com/guide/practices/ui_guidelines/icon_design.html
- https://developer.android.com/guide/practices/ui_guidelines/icon_design_launcher.html
- Android projects include a default icon
- Launcher icons named ic_launcher.png go into density specific res/mipmap folders (i.e. res/mipmap-mdpi)
- Launcher icons should be designed specifically for Android. Avoid mimicking visual elements and styles from other platforms.
- If you only include the a higher resolution version Android will generate the lower resolutions for you
- There are also required icons for the action bar, dialog & tab, notifications, and small contextual
- Tablets and other large screen devices request a launcher icon that is one density size larger than the device's actual density, so you should provide your launcher icon at the highest density possible.

Halloween
From the Welcome screen chose Start a new Android Studio Project (File | New | New Project)
Application Name: Halloween(Welcome)
Company name: a qualifier that will be appended to the package name
Package name: the fully qualified name for the project
Project location: the directory for your project /Users/aileen/Documents/AndroidProjects/Feelings
Form factors: Phone and Tablet (leave others unchecked)
Minimum SDK: API 19: Android 4.4 (KitKat)
Add an Activity to Mobile: Empty Activity
Activity Name: MainActivity (we can leave this)
Make sure Generate Layout File is checked
Layout name: activity_main
Backwards Compatibility should be checked

Open the activity_main.xml file in the Design editor.
The textView that is there should be in the center.
Remove **android:text** property for the textView either in the properties area or in the XML.
The Textview doesn't have an ID so add one so we can refer to this textview in our code.
android:id="@+id/message"
When you add IDs to resources they will have a + sign because you are creating the ID.
You don't use the + when you are referencing the resources.
We will use the ID when referring to it in our code.

Button
In Design mode add a button above the textView. You'll be able to see the textView when you drag over the view. Or switch to blueprint mode.
Look in the xml and see what was added.
Notice it has **android:id**

You'll also notice it has 2 errors. Look at what they say. We'll deal with the text first.
The button got created with default text, let's change that.
Add string resources in strings.xml
<string name="boo">Boo</string>
Back in activity_main.xml update the text
android:text="@string/boo"
The second error says that it's not constrained vertically. You can see it has left and right constraints for it's horizontal positioning.
In design view select the button and chose Infer Constraints which is the icon with 2 yellow plus signs. You can see this added a constraint for the bottom margin to the message textView with the value of 80dp.
Now we want the button to do something. In the button tag start typing onclick and use autocomplete.
android:onClick="sayBoo"
sayBoo is the method we want the button to call in our code when the click event fires.
Now we have to create this method in our mainActivity.java file
Either click on the lightbulb and chose Create sayBoo(View) in MainActivity or just go into MainActivity.java and create it.

Your method must follow the notation **public void methodname(View view) { }**
Android looks for a public method with a void return value, with a method name that matches the method specified in the layout XML.
The parameter refers to the GUI component that triggers the method (in this case, the button). Buttons and textviews are both of type View.

You need to import View and TextView so either add it manually or have AS add it automatically.
Android Studio | Preferences | Editor | General | Auto Import | Java and check "Add unambigious imports on the fly" and "Optimize imports on the fly"

```
import android.view.View;
import android.widget.TextView;

public void sayBoo(View view) {
    TextView booText = (TextView)findViewById(R.id.message);
    booText.setText("Happy Halloween!");
}
```

We create a TextView object called booText.
The findViewById(id) method is how Java can get access to the UI components using their ids.
The R.java class is automatically generated for us and keeps track of all our resources including ids.
R.id.message grabs a reference to our textview. (note that R must be capitalized)
So now our booText object is a reference to our textview.
We can set the text by using the setText() method.
Remember your semi colons!

Run it and try it out.
Can you make the text larger and the background a fun color?

EditText
Add an EditText (Plain Text) above the button.

It should have an id
android:id="@+id/editText"
Remove **android:text** as we don't want there to be text in the EditText
In strings.xml add <**string name="name"**>Name</**string**>
In activity_main.xml update the editView.
android:hint="@string/name"
There is still one error that it's not constrained vertically. Run it to see what happens if it's not constrained vertically. Then use Infer Constraints to add a vertical constraint.

Now let's update MainActivity.java so our message is personalized with our name. Update boo() after TextView.
EditText name = (EditText) findViewById(R.id.***editText***);
String nameValue = name.getText().toString();
booText.setText(**"Happy Halloween "** + nameValue + **" !"**);

We create a name object so we have a reference to our EditText.
Then we create a string and use getText() to get the text in the EditText and toString() to cast it to a String so we can use it in our TextView.
Use the + to concatenate strings and/or variable values.
Run your app.

In a larger app where you need access to the UI components throughout the class you can make them global by adding TextView booText; right under the class definition.

ImageView
Copy and paste your image into the drawables folder. (ghost.png)
Add an imageView below the textView.
If it makes you pick an image go ahead and chose ghost.png
Move it around so it fits. Notice how the constraint values automatically change.
Remove **app:srcCompat="@drawable/ghost"** so the app starts without an image, we'll assign it programmatically.

Add logic to MainActivity.java

ImageView ghost = (ImageView)findViewById(R.id.***imageView***);
ghost.setImageResource(R.drawable.***ghost***);

The R class uses the name of the drawable resource so make sure this matches the name you gave it in AS.

Move the widgets around in the design view, the constraints will update automatically.
Use Instant Run to quickly see the changes in the emulator.

Launcher icons
Add your own launcher icon to your app. (pumpkin.png)
Select the res folder right-click and select New > Image Asset
Or File > New > Image Asset.
Icon type: Launcher Icons
Name: leave as ic_launcher so you don't need to change it in the Android_manifest.xml file

(ic_launcher_round is needed for the Google Pixel)
Asset Type: Image
Browse to your image.
Trim - To adjust the margin between the icon graphic and border in the source asset, select Yes. This operation removes transparent space, while preserving the aspect ratio. To leave the source asset unchanged, select No. Default: No
Padding - If you want to adjust the source asset padding on all four sides, move the slider. Select a value between -10% and 50%. If you also select Trim, the trimming happens first. Default: 0%
Foreground - To change the foreground color for a Clip Art or Text icon, click the field. In the Select Color dialog, specify a color and then click Choose. The new value appears in the field. Default: 000000
Background - To change the background color, click the field. In the Select Color dialog, specify a color and then click Choose. The new value appears in the field. Default: FFFFFF
Scaling - To fit the icon size, select Crop or Shrink to Fit. With crop, the image edges can be cut off, and with shrink, they aren't. You can adjust the padding, if needed, if the source asset still doesn't fit well. Default: Shrink to Fit
Shape - To place a backdrop behind your source asset, select a shape, one of circle, square, vertical rectangle, or horizontal rectangle. **For a transparent backdrop, select None.** Default: Square
Effect - If you want to add a dog-ear effect to the upper right of a square or rectangle shape, select DogEar. Otherwise, select None. Default: None
Image Asset Studio places the icon within a transparent square so there's some padding on the edges. The padding provides adequate space for the standard drop-shadow icon effect.

You will see the warning that an icon with the same name exists. That's because Android Studio provides default launcher icons, we're replacing those so just ignore this warning.
Next
Make sure it's saving your icons to src/main/res and you'll see the various mipmap folders.
Finish

Now run your app and look at your launcher icon by clicking the right button, or go to home and then all apps to see it on your home screen.

You can download the Android Icon Templates Pack
https://developer.android.com/guide/practices/ui_guidelines/icon_design.html
You can also use Android Asset Studio to create all your launcher images.
http://romannurik.github.io/AndroidAssetStudio/index.html

Android Manifest
The manifest file is the central point of declaration for the project, including everything from activities and services, to permissions and SDK compatibility, and more.
If you look in the manifests/Android_manifest.xml file you will see an application label that holds the string app_name(hover to see the resource name). To change this go into the strings.xml file and change that resource to change your app name.
<**string name="app_name**">Welcome</**string**>
You'll also see the app theme **android:theme="@style/AppTheme"**
If you open styles.xml you'll see where the style name AppTheme is defined. You can change the parent to a theme available for the API levels you supported or customize your theme by changing the colors in the colors.xml file.