

Mobile Application Development

Week 9: Android Development Intro

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

Views

<https://developer.android.com/training/basics/firstapp/building-ui>

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/kotlin/android/view/View>
 - TextView
 - EditText
 - Button
 - ImageView
 - Check box
 - and many others
- Views that can contain other views are ViewGroups. They're 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.

TextView

- Text views are used to display text `<TextView .../>` **android.widget.TextView**
<https://developer.android.com/reference/kotlin/android/widget/TextView>

- The **android:textSize** attribute controls the size of the text
 - Use the sp unit for scale-independent pixels
 - Scales based on the user's font size setting
- The text property stores the String in the TextView
- Not editable by the user

EditText

- Edit text is like a text view but editable **<EditText .../> android.widget.EditText** <https://developer.android.com/reference/kotlin/android/widget/EditText>
- 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 "|"
- The text property stores the String in the EditText

Button

<https://developer.android.com/guide/topics/ui/controls/button.html>

- Buttons usually make your app do something when clicked **<Button .../> android.widget.Button** <https://developer.android.com/reference/kotlin/android/widget/Button>
- 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/kotlin/android/widget/ImageView>
- 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

R class

The R.java file acts as an index to all of the resources identified in your project. This file is automatically generated for every Android project and the "R" stands for resources. Any time you change, add, or remove a resource, the R class is automatically regenerated.

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 and they became the default layout when you create a new project starting in 2.3.3 so we will be using them. For now make sure AutoConnect is on (magnet icon) so it will handle most of our constraints for us.

Halloween

From the Welcome screen chose Start a new Android Studio Project (File | New | New Project)

Select a Project Template: In the Phone and Tablet tab pick Empty activity.

Name: Halloween

Package name: the fully qualified name for the project

Save location: the directory for your project (make sure there is a directory with the name of the project after the location)

Language: Kotlin

Minimum SDK: API 21: Android 5.0 Lollipop (21 is the minimum API for Material Design)

(Help me choose will show you the cumulative distribution of the API levels)

Leave legacy libraries unchecked (using these restricts usage of some newer capabilities)

Finish

Open the activity_main.xml file in the Design editor.

The textView that is there should be in the center.

Remove "Hello World" from **android:text** property for the textView so it's blank.

The Textview doesn't have an ID so add one so we can refer to this textview in our code.

Find the attribute id and give it a value of textMessage.

In the XML you'll see **android:id="@+id/textMessage"**

When you add IDs to views they will have a + sign in the XML because you are creating the ID.

You don't use the + when you are referencing the views.

We will use the ID when referring to the view in our code.

Button

In Design mode add a button above the textView.

In the Palette you'll see Button under Common as well as Buttons. Drag one out and place it 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 an id called "button".

You'll also notice it has 1 error and 1 warning. Look at what they say. We'll deal with the text first.

The button got created with default text which isn't recommended so let's change that.

Select the button and find the text property in the attributes pane.

Click the bar next to the text property

Click the + in the top left to add a string value.

(In the XML select the string and Option/Alt Enter will bring up a menu to extract string resource)

Resource name: buttonText

Resource value: Boo

Source set: main

File name strings.xml

Make sure the values directory is checked.

Now the text property should be set to **"@string/buttonText"**

The '@' sign means it's defined as a resource.

Design mode will show that our Button has the new string value.

The second error says that it's not constrained vertically. You can see it has left and right constraints for its horizontal positioning.

In design view select the button and chose Infer Constraints which is the icon that looks like a magic wand .

This adds a constraint for the bottom margin to the message textView with a value in dp (mine is 64).

Now we want the button to do something. Select the button tag and find the onclick attribute and set the value to 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.kt file

In the XML find `android:onClick="sayBoo"`

Click on sayBoo and a lightbulb should appear on the left.

Click on the lightbulb and chose Create sayBoo(View) in MainActivity.

Or just go into MainActivity.kt and create the method.

The method must follow the structure `fun methodName(view: View) { }`

Android looks for a method with a method name that matches the method specified in the layout file.

The parameter refers to the view 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.

1. Open the **Settings** or **Preferences** dialog:
 - On Windows or Linux, select **File > Settings** from the menu bar.
 - On macOS, select **Android Studio > Preferences** from the menu bar.
2. Navigate to Editor | General | Auto Import | Kotlin and check "Add unambiguous imports on the fly" and "Optimize imports on the fly"

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

fun sayBoo(view: View) {
    val booText = findViewById<TextView>(R.id.textMessage)
    booText.text = "Boo!"
}
```

val creates an immutable variable (a constant). var is used for mutable variables.

We create a TextView object called booText.

The findViewById(id) method is how Kotlin can get access to the views using their ids.

<TextView> defines what type of View will be returned.

The R.java class is automatically generated for us and keeps track of all our resources including ids.

R.id.textMessage grabs a reference to our textview. (note that R must be capitalized)

So now our booText object is a reference to our text view.

We can set the text through the text property.

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

EditText

Add an EditText (Plain Text) above the button. You can find it in the Palette under Text.

It should have an id, let's shorten it to editTextName.

Remove Name from the text property as don't want there to be text in the EditText.

But a hint would help indicate to the user what they should enter.

Find the hint attribute and click the bar next to it to access resources and add a string resource.

Resource name: editName

Resource value: Name

Source set: main

File name strings.xml

Make sure the values directory is checked.

Now the hint property should be set to **"@string/editName"**

The ems property might have a value, that determines the width of the text field

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 (magic wand) and it will add a vertical constraint.

Now let's update MainActivity.kt so our message is personalized with our name. Update sayBoo().

```
val editName = findViewById<EditText>(R.id.editTextName)
val name = editName.text
```

We create an editName object so we have a reference to our EditText.

We store the value from its text property in a variable called name.

```
booText.setText("Happy Halloween " + name + " !");
```

Use the + to concatenate strings and/or variable values.

Run your app.

Can you figure out how to center the text in the TextView?

```
android:gravity="center"
```

ImageView

Drag and drop or copy and paste your image into the drawables folder. (ghost.png)

In activity_main.xml add an imageView below the textView by dragging it out from the Palette.

If it makes you pick an image go ahead and chose ghost.png in the project section.

Move it around so it fits. Notice how the constraint values automatically change. You can also easily change the value for a constraint in the attributes panel.

Notice that it has the id imageView.

It also has an error for missing a vertical constraint. Select the imageView and click Infer Constraints for a vertical constraint to be added.

Once you have it positioned, remove @drawable/ghost from the srcCompat property so the app starts without an image, we'll assign it programmatically.

Update sayBoo() in MainActivity.kt so our image shows up when the button is tapped.

```
val imageGhost = findViewById<ImageView>(R.id.imageView)
imageGhost.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.

In an app where you need access to the UI components throughout the class you can make them global by defining them at the class level. But you can't call findViewById() until after the layout file has been set as the content view. So you'll need the keyword lateinit before it since it will be initialized later.



```
lateinit var editName: EditText
```

A good place to initialize it is in onCreate() after setContentView().

```
editName = findViewById<EditText>(R.id.editTextName)
```

Apply Changes

To run your app again in the emulator you have a few choices.

- **Run** deploys all changes and restart the application.
 - Use this option when the changes that you have made cannot be applied using either of the Apply Changes options.
- **Apply Changes and Restart Activity**  will attempt to apply your resource and code changes and restart only your activity without restarting your app.
 - Generally, you can use this option when you've modified code in the body of a method or modified an existing resource.
- **Apply Code Changes**  will attempt to apply only your code changes without restarting anything.
 - Generally, you can use this option when you've modified code in the body of a method but you have not modified any resources.

Enable Run fallback for Apply Changes

After you've clicked either **Apply Changes and Restart Activity** or **Apply Code Changes**, Android Studio builds a new APK and determines whether the changes can be applied. If the changes can't be applied and would cause Apply Changes to fail, Android Studio prompts you to Run your app again instead. However, if you don't want to be prompted every time this occurs, you can configure Android Studio to automatically rerun your app when changes can't be applied.

To enable this behavior, follow these steps:

3. Open the **Settings** or **Preferences** dialog:
 - On Windows or Linux, select **File > Settings** from the menu bar.
 - On macOS, select **Android Studio > Preferences** from the menu bar.
4. Navigate to **Build, Execution, Deployment > Deployment**.
5. Select the checkboxes to enable automatic Run fallback for either of the Apply Changes actions.
6. Click **OK**.