COSC 326 – Android Development
Lab 1 – Winter 2023

In this lab, we'll get our first introduction to working with ***Android Studio***, a development IDE which includes both programming and UI design tools for working with Android applications. We'll create a new project with a simple ***activity***, check out the ***manifest*** file, and then ***build*** our application using the built-in Gradle toolchain and test it on some emulated Android hardware.

Getting Android Studio and Setting Up a New Project

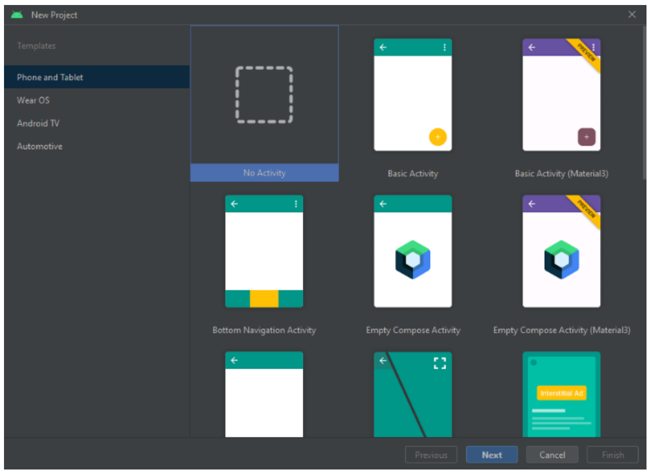
You can download ***Android Studio*** from the following link. It is available for Windows, Mac and Linux:

https://developer.android.com/studio

Some notes to help make sure your Android Studio install goes as smoothly as possible:

- Make sure your ***C:*** drive, or wherever your ***user*** directory is has at least around ***10GB*** of storage available. Even if Android Studio is installed on a different drive, it's ***emulators*** will still be installed in your user directory. Failure to allocate enough space will cause the emulator to fail during installation, or while booting the VM.
- You may want to install Android Studio in a separate location or a separate drive. There is an option for this during the install. Note that the ***final installation size*** will be larger than the initial download as a number of files are downloaded and unpackaged during the install process.
- For comparison sake, on a machine with a five year old Intel i3 CPU and 8GB of RAM, my emulation speed is sluggish, but not unbearable. Keep this in mind if you are trying to run Android Studio on a machine with very poor specs.

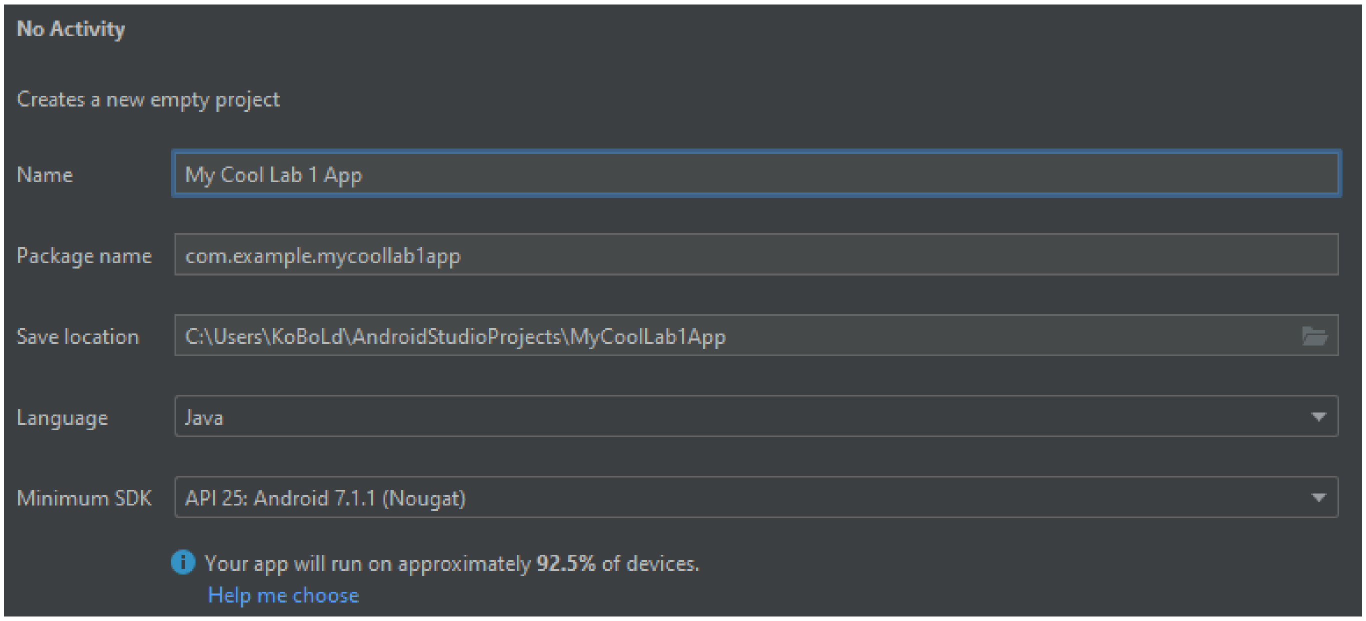
Once you've followed through the installation process and Android Studio is installed, open it and ***create a new project*** (under the ***file*** > ***new*** > ***new project*** menu). This will present you with a choice of starting templates:



From this menu, select the ***Phone or Tablet*** menu, and then select the ***No Activity*** option. This will create a completely empty Android project, which consists of just a ***manifest*** file.

We will be using some of the other templates in other labs. They come with activities and layouts already set up, to let you quickly build apps with common elements. We'll do this manually to start with.

Once you've selected the empty project template, you'll be prompted to enter an application name and a ***minimum SDK version***. The minimum version is used to specify what the absolute minimum version of Android required to run your application is. It will give you a rough percentage of devices that your app will be compatible with, based on the SDK version you choose. You may select a higher SDK version, but I recommend leaving it at ***Android 7.1.1 (Nougat)***. You may name your app whatever you want, but make sure you check that the language is set to Java, ***not Kotlin!*** An example is given below:



No Activity

Creates a new empty project

Name: My Cool Lab 1 App

Package name: com.example.mycoollab1app

Save location: C:\Users\KoBoLd\AndroidStudioProjects\MyCoolLab1App

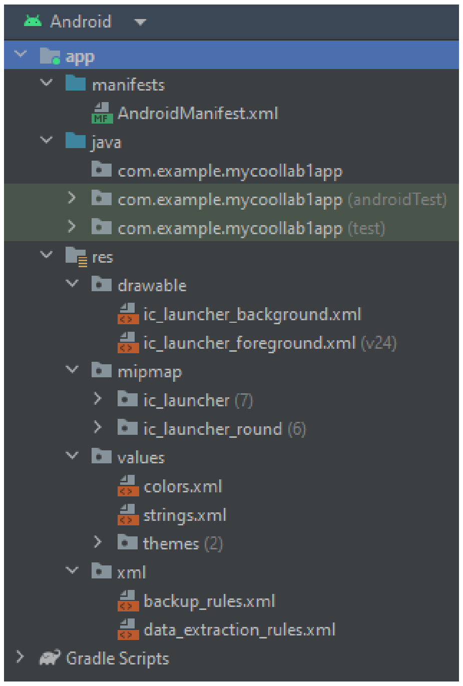
Language: Java

Minimum SDK: API 25: Android 7.1.1 (Nougat)

Your app will run on approximately **92.5%** of devices.

Help me choose

Click finish to create the project. It may take some time to finish this, as Android Studio uses Gradle to build it's projects and it'll take a while to set everything up. Once it's all set up, you should be able to expand the ***project explorer tree*** on the left hand side and see the folders and files that make up your project.



```
Android
app
  manifests
    AndroidManifest.xml
  java
    com.example.mycoollab1app
    com.example.mycoollab1app (androidTest)
    com.example.mycoollab1app (test)
  res
    drawable
      ic_launcher_background.xml
      ic_launcher_foreground.xml (v24)
    mipmap
      ic_launcher (7)
      ic_launcher_round (6)
    values
      colors.xml
      strings.xml
      themes (2)
    xml
      backup_rules.xml
      data_extraction_rules.xml
Gradle Scripts
```

First, go to the ***manifests folder***, and double click on the ***AndroidManifest.xml*** file. This is the main application manifest, where all your activities, services, and other components will be listed. At the moment, it mostly just includes some placeholders:

- The displayed application name (***android:label***)
- A default icon (***android:icon***, which points to default values in ***res/mipmap/ic_launcher***)
- Some default rules for data backup and what data may be modified or touched by the user and/or other applications (the ***dataExtractionRules*** and ***fullBackupContent*** sections)

This ***manifest*** file should automatically update as we build our application, so remember what it looks like now – it'll look different later.

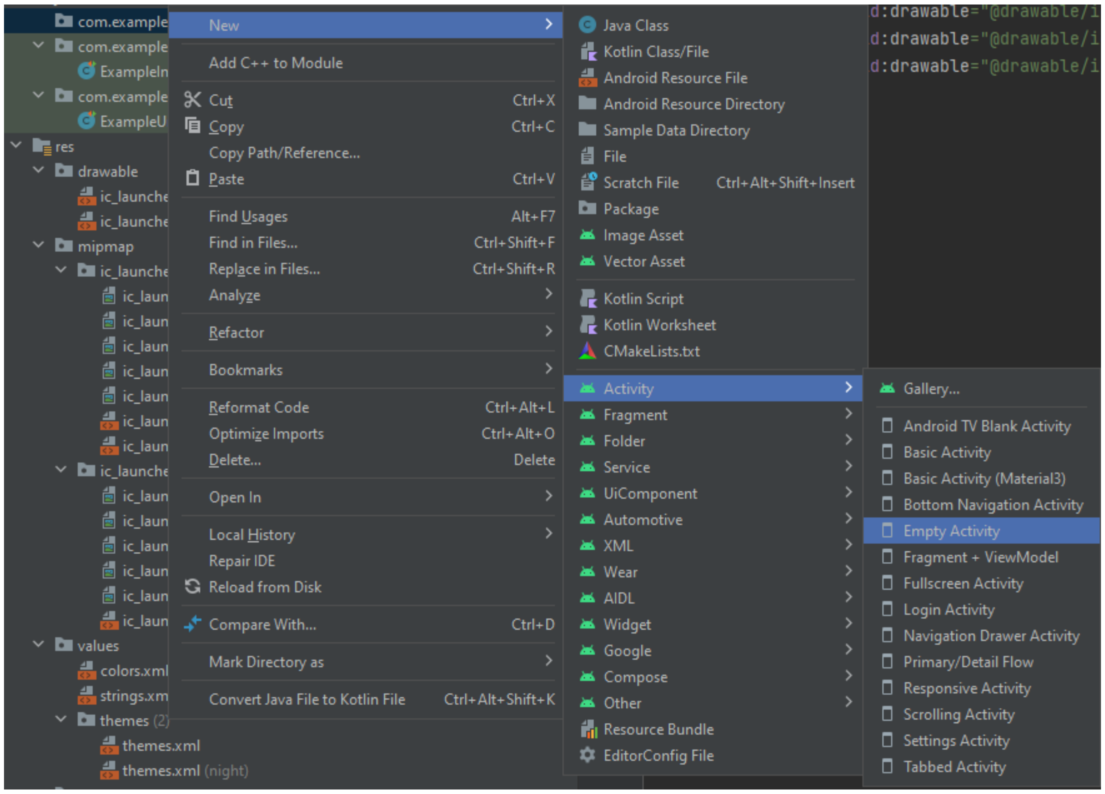
The ***java*** folders are where our code will go – at the moment, it just includes some default test files. When we create activities, they will be placed into the main package in the java folder.

The ***res*** folders hold ***resources*** four our application. For the most part, these are currently default XML files that specify things like default colors and themes, as well as those default Android test icons. Note that the icons are in ***webp*** format in a variety of sizes, but are described using an ***XML*** file that states which ones are used (for example, the icon to use when the app is running in the background, versus when it's running in the foreground).

All of these folders will have files added to them automatically as we build our application, so it's good to understand what an "empty" project looks like beforehand.

<u>Setting Up Our First Activity</u>

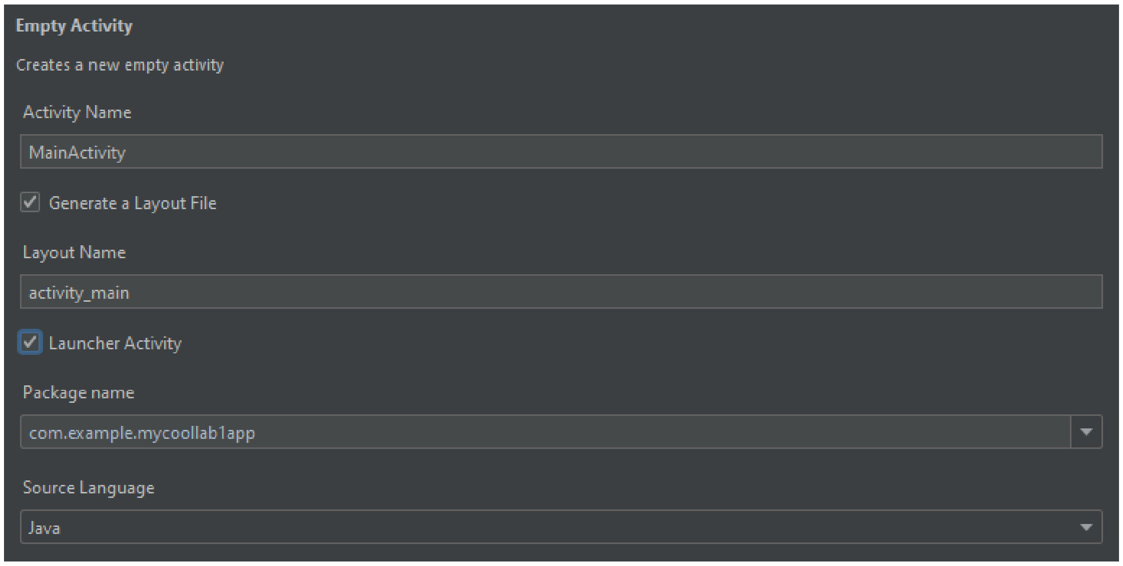
One of the reasons for using ***Android Studio*** is that it makes creating new activities and components for our application very easy. Just ***right-click*** anywhere in the ***project explorer***, and select ***New***, then ***Activity***, and select the ***Empty Activity*** option:



```
d:drawable="@drawable/i
d:drawable="@drawable/i
d:drawable="@drawable/i
```

This will create a new empty ***activity***, including the Java class file for the activity, the required ***XML layout file*** for the UI, and the entry in your ***app manifest*** file for this new activity.

You can leave the settings of the empty activity alone – the only setting you need to alter is the ***Launcher Activity*** setting. Make sure this box is ***checked***. The launcher activity tells Android that this activity is one that can be used as an "entrypoint" to your application – i.e. where you go when the app first starts up.


Empty Activity

Creates a new empty activity

Activity Name: MainActivity

[✓] Generate a Layout File

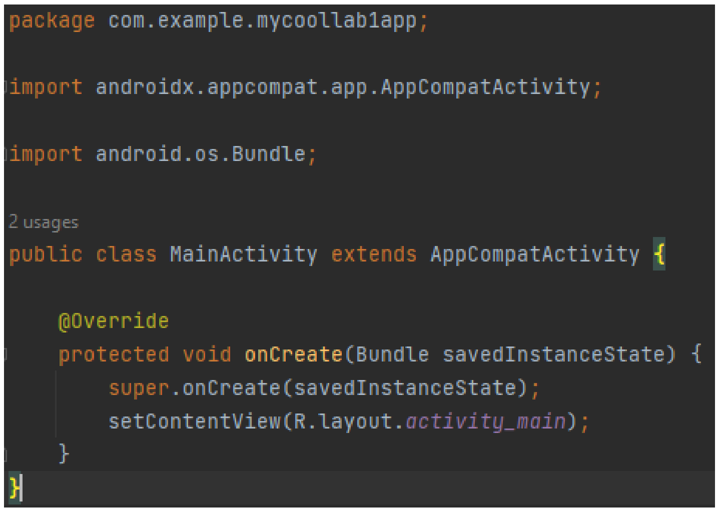
Layout Name: activity_main

[✓] Launcher Activity

Package name: com.example.mycoollab1app

Source Language: Java

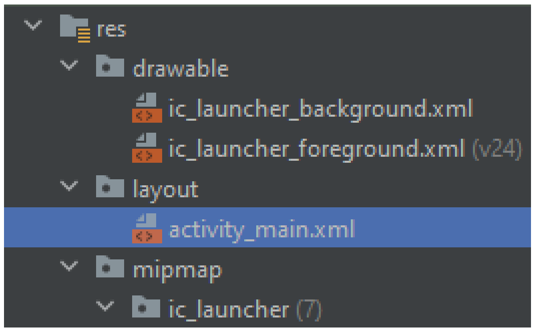
Once we've created the activity, and Gradle is finished syncing everything up, try opening the new activity java file (under the ***java*** folder). It should be very short, and look like the code shown here.


```
package com.example.mycoollab1app;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;

2 usages
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

The activity class inherits from the ***AppCompatActivity*** class. It has only one method right now: ***onCreate***, which overrides the class definition of the method. This method specifies which ***layout*** to use for the activity.

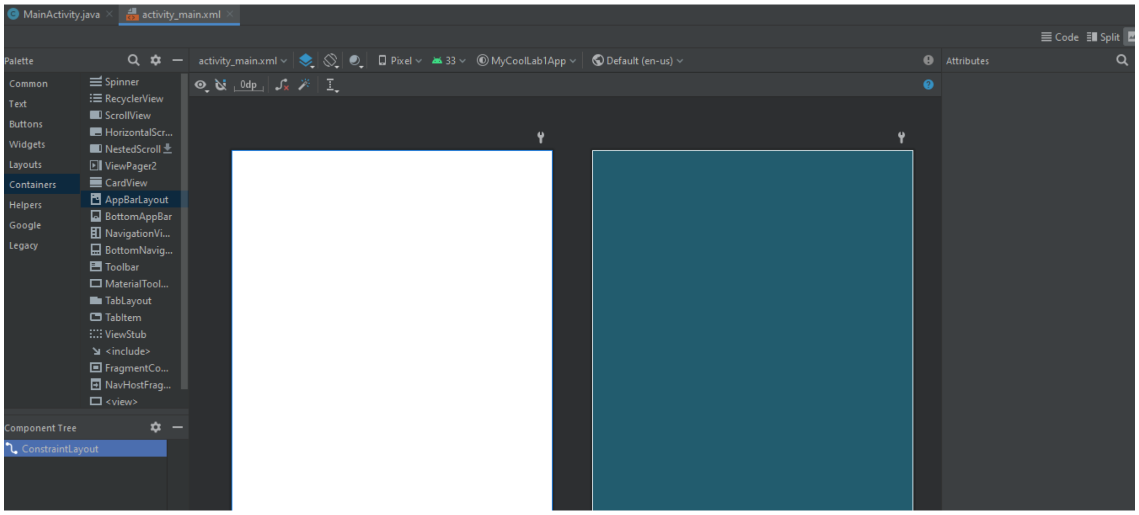
The ***layout*** is used for dealing with the actual layout and design of the UI, while the Java code of the activity is reserved for dealing with interaction, storage, and listening for input. This helps keep the UI and design side of the application separate from the backend.

The layout that was created can be found in the ***res*** folder, under the ***layouts*** subfolder. In this case, it's named ***activity_main.xml***. Go ahead and open this file in Android Studio (double-click on the filename). This will open the Android Studio Layout Editor.


```
res
  drawable
    ic_launcher_background.xml
    ic_launcher_foreground.xml (v24)
  layout
    activity_main.xml
  mipmap
    ic_launcher (7)
```

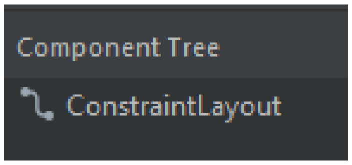
Using the Layout Editor

Let's create a layout for our new activity. The Layout Editor in Android Studio looks like the following:



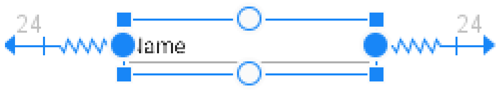
The interface of the Layout Editor includes:

- A ***palette*** of various elements that can be included in the activity, ranging from layouts and containers (means of organizing elements) to individual text areas, input fields, or buttons.
- A ***component tree***, which describes the XML hierarchy of the GUI components (i.e. a button may be a child of a container).
- Two ***views***: a ***rendered view*** (left) and a ***blueprint view*** (right). The rendered view shows your activity as it rendered, while the blueprint shows you outlines of the components in the activity.
- ***Attributes*** of a selected component will be shown on the right hand side of the editor.
- At the top, some option are available. Specifically, the ***phone icon*** will allow you to select different phone screen sizes, tablet sizes, and even TVs or wearables to see how your GUI layout will render.



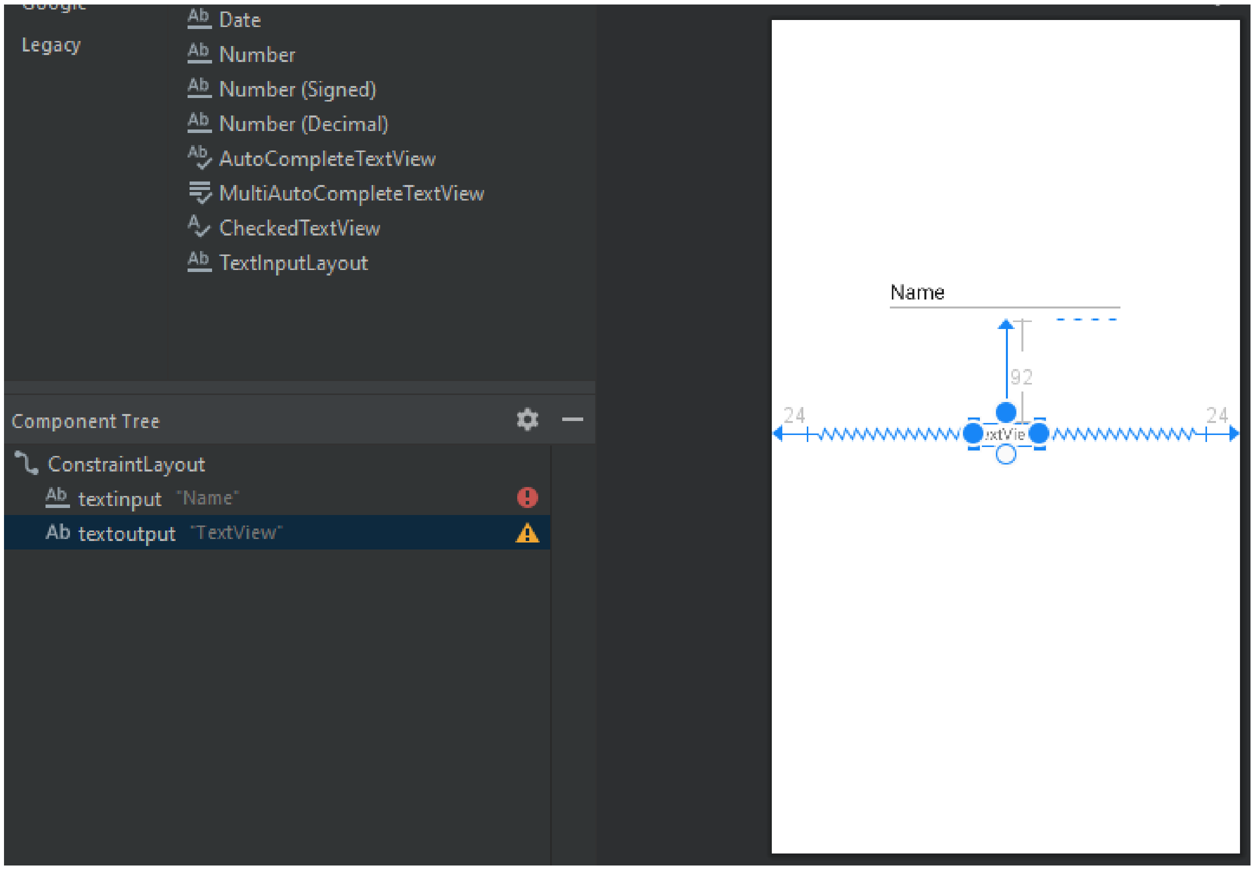
To start with, our activity will only contain a ***ConstraintLayout*** object. This allows us to "float" elements around the page based on constraints (i.e., keep this object X distance from the left, right, above, or below – like padding or margins).

Let's add a text input to our constraint layout. From the ***palette,*** look for the ***Plaintext*** element – this is a plain text user input. Go ahead and drag it onto the design area, or right click and select ***Add to Design***. Once it is in your design, open it's ***Attributes*** and check under the ***Layout*** section – set the plaintext element to have constrains on both sides, so that it floats in the middle of the application page.



At the same time, add a vertical constraint to the text input. change the ***id*** of the plaintext input to be ***textinput***. We'll reference this in our activity's Java code. Then, do the same thing with a ***TextView*** element – add it to your layout, give it some constraints, and then give it the id ***textoutput***.

When you're done, your activity should look something like this:

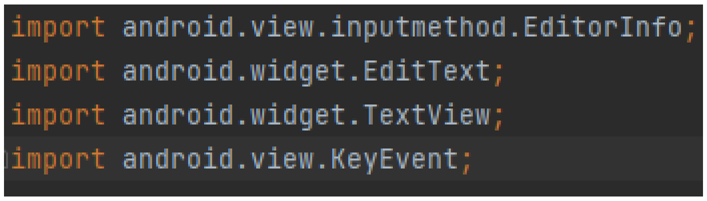


Once we've gotten some components placed into our layout and named accordingly, we're ready to move over to the ***Java*** side of things start giving our layout some backend logic.

<u>The Activity Class</u>

First, go ahead and open up the ***MainActivity*** Java class file which was shown in an earlier image. This is a very barebones skeleton for the application, it doesn't even have the necessary imports to interact with our layout at this stage. This will be the first thing we add.

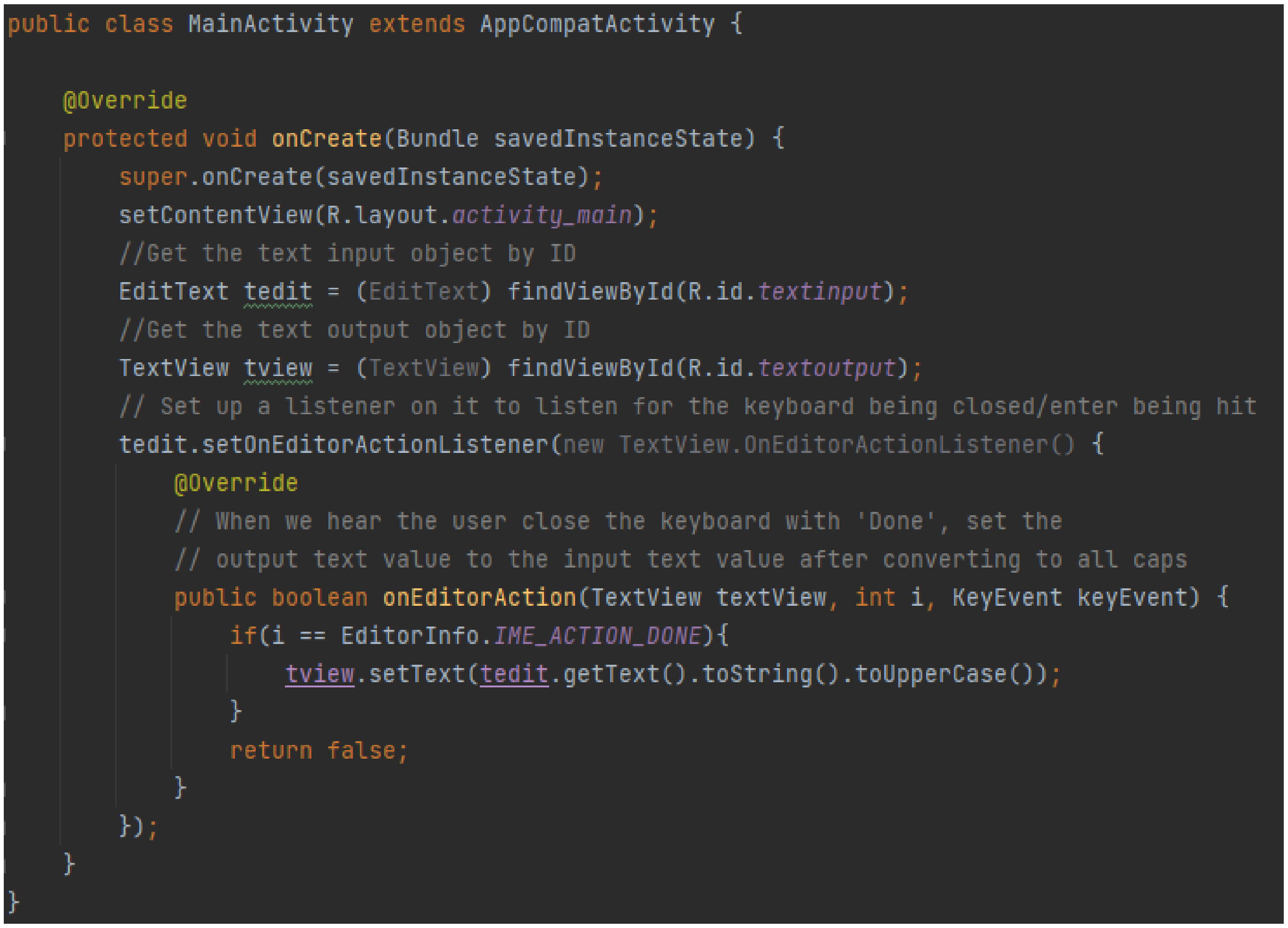
We're going to import some packages – the ***EditText*** and ***TextView*** packages to handle our UI elements, ***android.view.KeyEvent*** to listen for the user entering a key, and



```
import android.view.inputmethod.EditorInfo;
import android.widget.EditText;
import android.widget.TextView;
import android.view.KeyEvent;
```

inputmethod.EditorInfo to fetch the current editor (i.e. Android keyboard)'s last action (i.e. entering or submitting our text).

Our application will be very simple: We'll add a ***listener*** to our ***EditText*** input, which will listen for when the user enters the 'Done' key on the Android keyboard. When this happens, we'll trigger some code: we'll take the text, cast it to a string, convert it to all uppercase (like you could in any Java implementation), and then we'll write the contents to the ***Text*** property of our ***TextView*** object. The code, with some comments, is provided below.



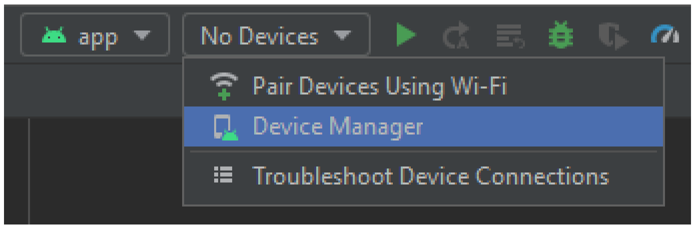
```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        //Get the text input object by ID
        EditText tedit = (EditText) findViewById(R.id.textinput);
        //Get the text output object by ID
        TextView tview = (TextView) findViewById(R.id.textoutput);
        // Set up a listener on it to listen for the keyboard being closed/enter being hit
        tedit.setOnEditorActionListener(new TextView.OnEditorActionListener() {
            @Override
            // When we hear the user close the keyboard with 'Done', set the
            // output text value to the input text value after converting to all caps
            public boolean onEditorAction(TextView textView, int i, KeyEvent keyEvent) {
                if(i == EditorInfo.IME_ACTION_DONE){
                    tview.setText(tedit.getText().toString().toUpperCase());
                }
                return false;
            }
        });
    }
}
```

Note that Android Studio will ***attempt*** to include the correct imports for unimported packages, but due to many overlapping package names between ***Java*** and the ***Android API***, it may be unable to find them, or import the wrong package. It's probably best for now to manually provide the imports.

Once you've modified the ***MainActivity*** Java file, we're just about ready to try our first round of ***emulation*** – and we'll get to see if the code actually works.

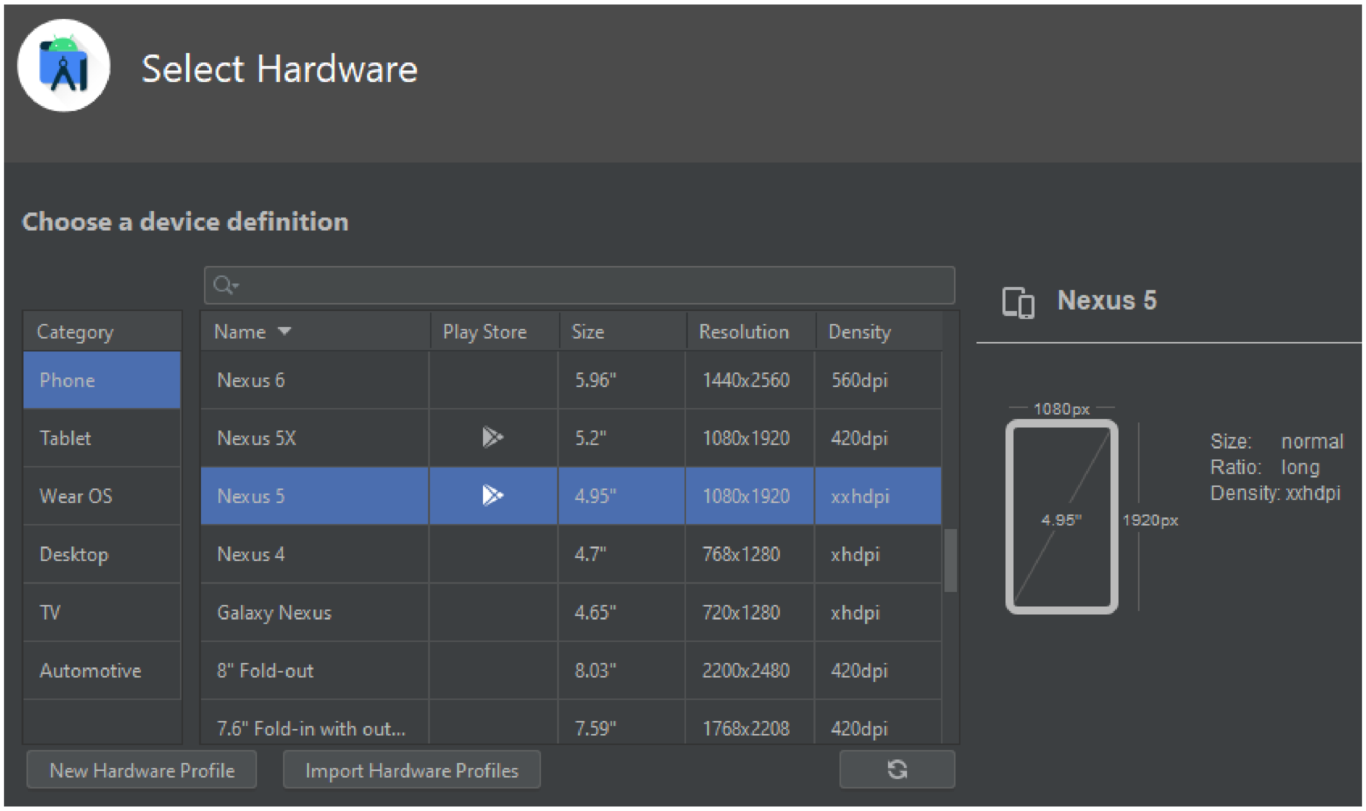
Setting Up Devices

This is the portion of the lab that is most likely to give you trouble due to ***underlying system configuration issues.*** As previously mentioned, make sure that your ***C:/ drive, or wherever your user folder is, has at least 10GB of space available.*** *You will get errors otherwise.*



In the top menu next to the ***run*** button, you'll see a dropdown that should default to saying "***No devices***". Go ahead and click on this, and then select the ***Device Manager*** from the dropdown menu as shown on the left.

The ***device manager*** will present you with a number of different options for emulating Android devices, ranging from phones and tablets to desktops (i.e. ChromeOS) and televisions. ***If you already own an Android phone***, check the dropdown to see if it's available for emulation by default. Being able to emulate the same device you would like to test your applications on will be very useful. If your phone isn't available by name, you should be able to find a generic device profile with similar resolution and screen size to use instead.



Category	Name	Play Store	Size	Resolution	Density
Phone	Nexus 6		5.96"	1440x2560	560dpi
Tablet	Nexus 5X	▶	5.2"	1080x1920	420dpi
Wear OS	Nexus 5	▶	4.95"	1080x1920	xxhdpi
Desktop	Nexus 4		4.7"	768x1280	xhdpi
TV	Galaxy Nexus		4.65"	720x1280	xhdpi
Automotive	8" Fold-out		8.03"	2200x2480	420dpi
	7.6" Fold-in with out...		7.59"	1768x2208	420dpi

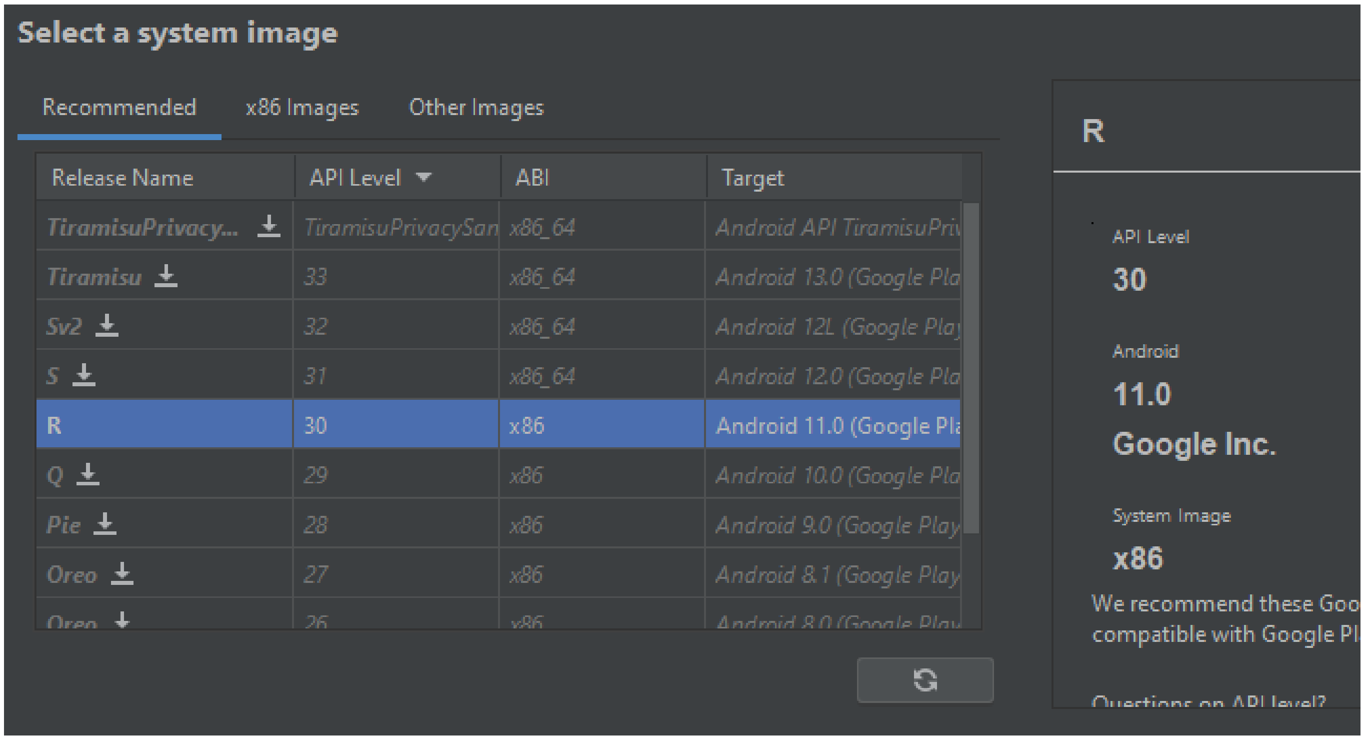
Nexus 5

1080px
4.95"
1920px

Size: normal
Ratio: long
Density: xxhdpi

Some devices have an icon in the ***Play Store*** column. This means that the device can support the Google Play store within the emulator. This isn't important for this lab, but may be important in the future if you need to download another app ***inside the emulation*** which your app will in turn work with (for example, an extension of an existing app on the marketplace).

Once we have selected the ***device***, we need to select a suitable ***system image*** for it. This will specify the exact version of the Android API that we are emulation. The recommended emulation option is normally good, unless you're specifically testing code for the latest platforms (or very outdated ones). In this case, it will likely recommend release R, API level 30, and Android 11.0. Since Android has a lot of backwards compatibility, even newer images (like this one) should be able to run apps designed for an older minimum environment (like Android 7.0).


Select a system image

Release Name	API Level	ABI	Target
TiramisuPrivacy...	TiramisuPrivacySan	x86_64	Android API TiramisuPriv
Tiramisu	33	x86_64	Android 13.0 (Google Pla
Sv2	32	x86_64	Android 12L (Google Play
S	31	x86_64	Android 12.0 (Google Pla
R	30	x86	Android 11.0 (Google Pla
Q	29	x86	Android 10.0 (Google Pla
Pie	28	x86	Android 9.0 (Google Play
Oreo	27	x86	Android 8.1 (Google Play
Oreo	26	x86	Android 8.0 (Google Play

R

API Level
30

Android
11.0
Google Inc.

System Image
x86

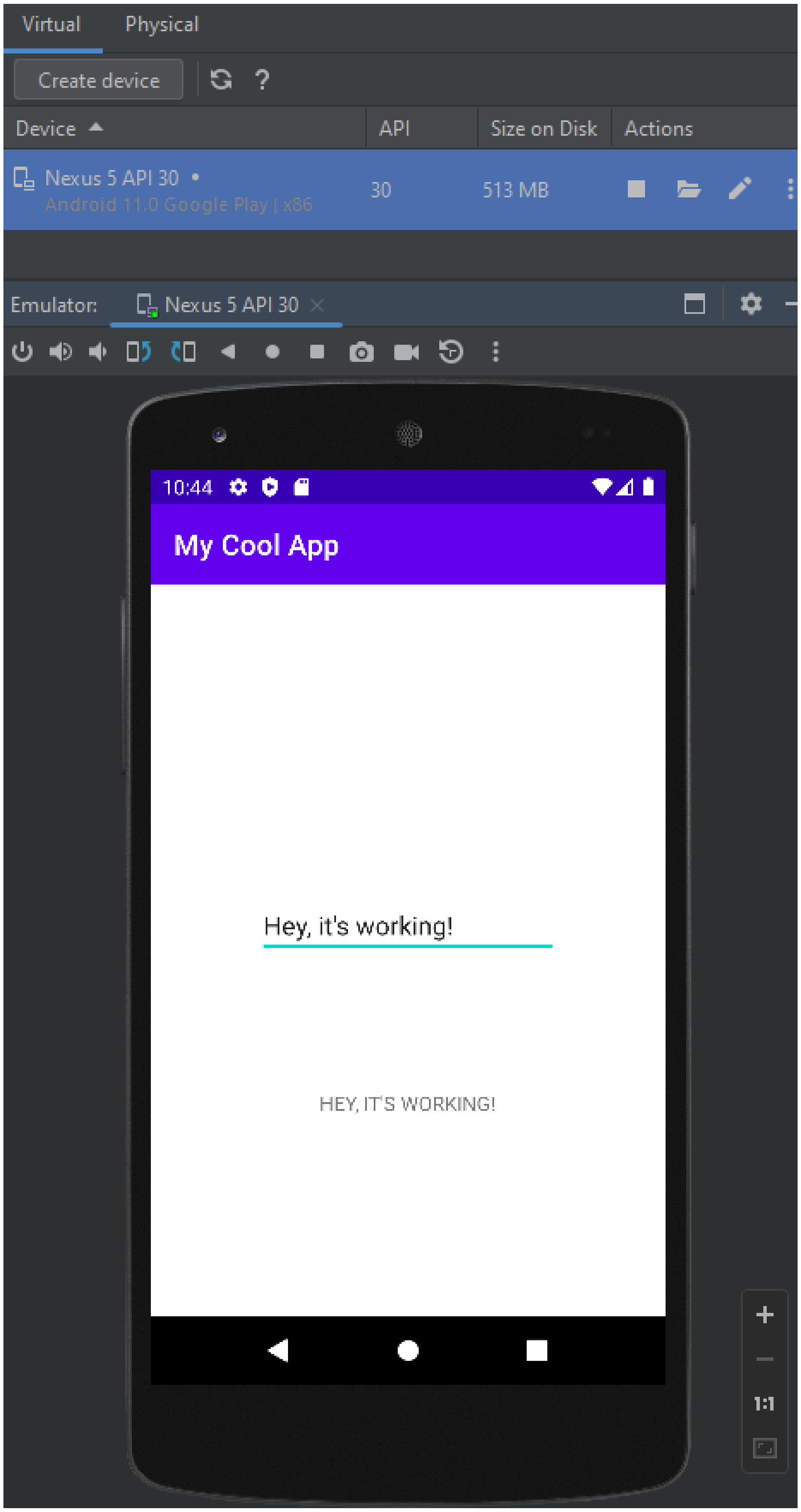
We recommend these Goo
compatible with Google Pl

Questions on API level?

First, select the appropriate system image you want to use, and then click the small ***download icon*** next to it. This should begin installing the system image. ***Be warned: these images are large! At least one gigabyte each typically***. So you may want to make sure you have plenty of space before you try downloading a bunch of system images.

Once you have a ***device*** and ***system image*** chosen, you're ready to try emulating your application. Click the ***run*** button. It may take some time for Gradle to finish building your application and the emulator to boot up, but once it is started, you should see something like the image on the next page.

This emulator supports most actions available in an actual hardware phone. You can open and close your application, provide input, and even use emulated versions of interfaces (like cameras, sensors, or GPS). Go ahead and try providing your application with input by clicking on the text input and typing something. Click the ***done*** button on the keyboard, and verify that your text input is now shown in lowercase below the text input, as shown in the image.



Marking

This lab is primarily about getting people into Android Studio and making sure that everyone is able to get the emulator running with a simple application. For a lab submission, just take a screenshot of your final emulated app output showing your name as the text input. Once you've finished, upload your screenshot on ***Moodle***.