

## SUBSYSTEM 4

---

### Writing Smart Phone Apps

# Table of Contents

Introduction.....	1
Objectives .....	2
Section 1: The Anatomy of an App .....	3
Application Lifecycle.....	4
Application States .....	4
Section 2: Android Studio .....	8
App Idea: What do I weigh on Jupiter? .....	11
Starting a Project .....	11
Section 3: Simulate and Run .....	24
Create a Virtual Device .....	24
Section 4: Adding Functionality .....	33
Create the User Interface.....	33
Write Code to Make Things Work .....	40
Add the other planets.....	53
Create a Custom Color .....	55
Add Padding to an Object.....	60
Adding a Custom Icon .....	61
Appendix A: Additional Resources.....	64
Appendix B: Loading an App on your Android Device .....	65
Appendix C: Code Listings .....	67

---

## Introduction

It is probably hard to imagine life before our constantly connected existence we have today. Cell phones started as a way to talk to people, advanced to a way to text people, and have come to the point that they act as our connected computers on the go. We can hook into the internet now wherever we are and do all of the things that we would normally do on a full sized computer. We can listen to streaming music while navigating to a restaurant we found while searching for local things to do.

One of the most impressive things our phones so is give us access to seemingly endless apps to satisfy our need to conduct business, catch up on shows we missed, or play games while we wait for our teeth cleaning. These apps have made the cell phone a one stop shop for so many things.

And yet, there is still more room to grow. There are people out there waiting for YOUR app. Writing apps is not only limited to software engineers. Today, anyone can write an app and post it for sale on the popular app markets. This subsystem will introduce the topic of developing apps for the Android operating system using Google's Android Studio. Android Studio will run on either Windows or Apple computers and is free.

## Objectives

After completion of this unit, the student will:

1. be able to explain the basics of phone app development.
2. be familiar with Integrated Development Environments and how they make software writing easier.
3. describe the basic framework of an Android App.
4. write, compile, and run Android programming code.
5. understand some of the basics of the Java programming language.
6. understand how to setup and run programs on a phone simulator.
7. understand the requirements to post an app on the app store.

Many people have dived into the world of app development. Some have been so excited, they have gone on to form companies and have made a good living writing apps. The skills gained in writing your own apps, as well as the understanding of how apps work, will lead you to a better understanding of computers and software in general.

Let's get started.

## Section 1: The Anatomy of an App

Android is an **operating system** developed by Google for the mobile market. An operating system is the fundamental underlying software that allows programs (apps) to communicate with hardware, memory, and manage computer resources. Programs cannot run on a machine without an operating system. Some examples of operating systems are Windows, Linux, macOS, UNIX, and Chrome OS. These software packages run on desktop computers, laptops, tablets, and mobile smartphones. Without them, every program would have to be tailored to the specifics of a particular hardware device. With these operating systems, we can create common interfaces so maybe the program just needs to request the device to play a song. The program would request it from the operating system and the operating system would go through the required steps to play it. I can now run this program on another platform and when the program requests to play the song, that new operating system will play the music on that particular device. By doing this, developers can write more generic programs and let the operating system handle the dirty work.

This idea is used extensively in Android programming. Android apps are programmed in **Java**. Java is a programming language that was envisioned as a “write once, run anywhere” language. That means you can write a program in Java, and then run it on multiple phones and tablets without having to rewrite the program code.

One of the things you need to understand about an Android app is the app lifecycle.

## Application Lifecycle

Because an Android app will run on mobile devices, it has a couple of specific requirements that most applications don't have to worry about. First, when you start an app on your phone, it fills the whole screen. You interact with it by touchscreen and button inputs. As you are doing this, your phone may ring and your phone app suddenly appears and covers the app you were on. When your call is done, you close your phone app and the original app reappears. Android needs to handle the fact that the current app may change states at any time. You may be using it, and then, load an app over it. You may then switch back to it later. If the phone had unlimited memory, this wouldn't be an issue. But the Android operating system makes sure it can always respond to your requests by controlling what apps are in memory and which ones get shut off. This would all be very complicated if Android didn't come to the rescue with a set of functions that are called throughout the lifespan of your app. These functions can be used to place the app and data in safe conditions and then restore the app when needed. Below is a discussion of these functions.

**Function:** A function is a set of programming instructions that are called to execute by a program or operating system. In the functions below, these are called by the Android operating system at specific times in the app lifecycle.

Android handles apps in the form of what are known as **Activities**. When you load a screen that interacts with the user, that is an activity. If you pop up a selection dialog, that is another activity. Each one of these activities will have its own lifecycle.

## Application States

An Android activity can exist in one of the following states:

**Active (or running):** The activity is in the foreground on the screen. It is the one that is active and taking input from the user.

**Paused:** The activity is visible, but something is covering it. This happens when a pop up message or dialog is open (it is also an activity) and is partially covering the other activity. A paused activity is still running in the background.

**Stopped:** If the activity is totally covered by another activity, the activity is stopped. In this state, the app is essentially frozen. It does not continue to run, but it has the ability to start running again where it left off. However, in this state it may be shut down completely if the operating system decides it needs the memory for higher priority tasks.

**Destroyed:** When an app is paused or stopped, the operating system can completely remove it from active memory. When the app starts again, it does not have the information of the state it was in when it was destroyed. So if you were working on a text message, and you shift to other apps to perform work, if the operating system destroys your app, you will lose the text you were working on.

Fortunately, by design Android gives you the opportunity to run code when it places your app in the various states. That means you can be notified when your app is about to be shut down and you can run some code to save progress in a game, save the text you were writing, log usage statistics, or anything else you want your app to do.

Here is a description of the functions Android uses at the various states:

**onCreate():** This function is called when the activity is first created. This is where you will do all of your setup for the application. You will assign variables, build your graphic interface with the user, initialize data, etc.

**onStart():** This function is called when the activity is just becoming visible to the user.

**onResume():** This function is called when your activity is ready to receive input from the user.

**onPause():** This function is called when another activity comes to the foreground of the screen. This is a good function to place code to save the

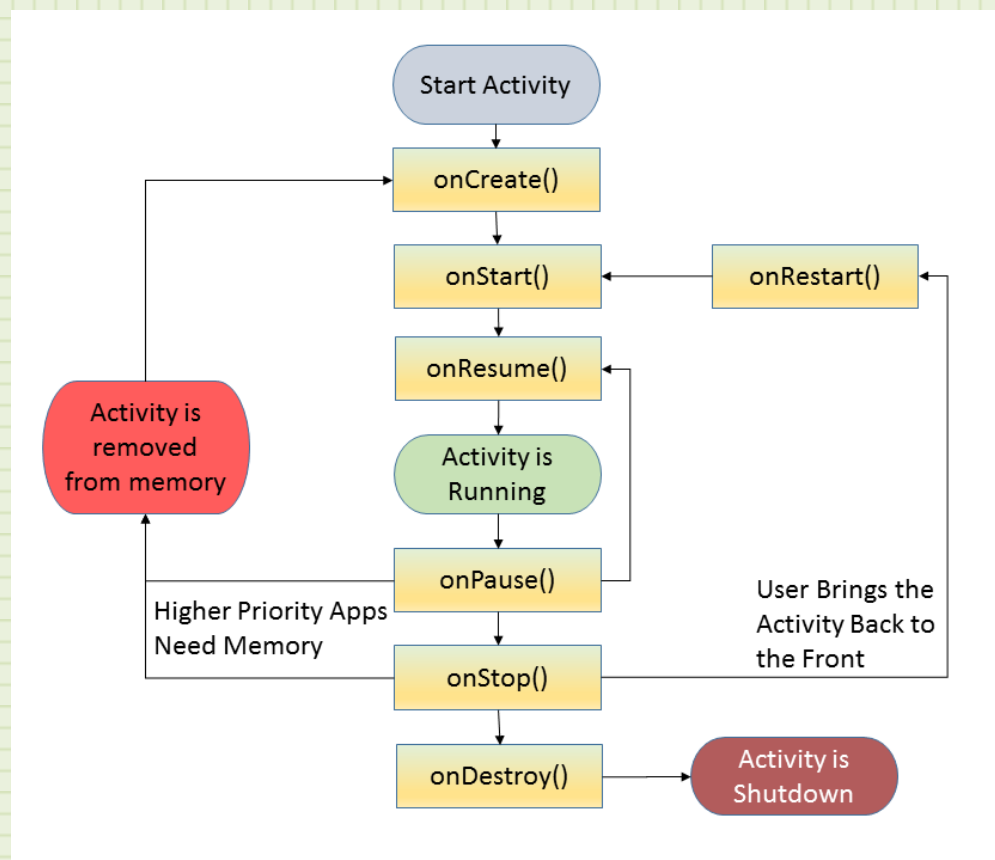
current state, stop running services and animations to conserve battery power, and other things to get the app ready to become dormant.

**onStop():** This function is called when the activity becomes completely covered by another activity and is no longer visible on the screen.

**onRestart():** This function is called when the user navigates back to the stopped activity to display it again. This function is always immediately followed by onStart().

**onDestroy():** This function is called when the activity is going to be unloaded from active memory. This can be because the program itself calls this function to destroy it or the operating system calls it to free up memory for other activities.

The following chart shows the flow of these states from one to another.





Most of the setup and shutdown code for an app is located in these functions. Most of the code to actually interface with the user and perform desired operations occurs in the box labeled “Activity is Running.” For a simple app, you may never need to add code to a lot of these functions. But you will always need to write some code in the onCreate() function.

Android apps are written by working on three separate parts of the app and then combining them together. Complicated apps may have many different parts, but all apps will have the following sections:

- 1) **The Manifest.** This is a file that hold information about the app itself. It has code to register activities with the operating system, set the minimum Android version, give the app permission to utilize your phones services and more.
- 2) **The Layout (User Interface or UI).** This is a file that looks a lot like the html code for a webpage. It establishes the user controls and displays that will be used in the app, determines their position on the screen and whether they are visible or not, what colors and fonts are used, etc. Android Studio gives you the ability to drag and drop components and write this code automatically.
- 3) **Java Files.** These files are where we write the code for the app. Here we will get the app to load our UI and recognize components, handle events like swipes, button pushes, and closing the app, and do the majority of the functionality that our app is intended to do.

With that foundation laid, let’s load Android Studio and start looking at how we can turn an idea into an app.

# Section

# 2

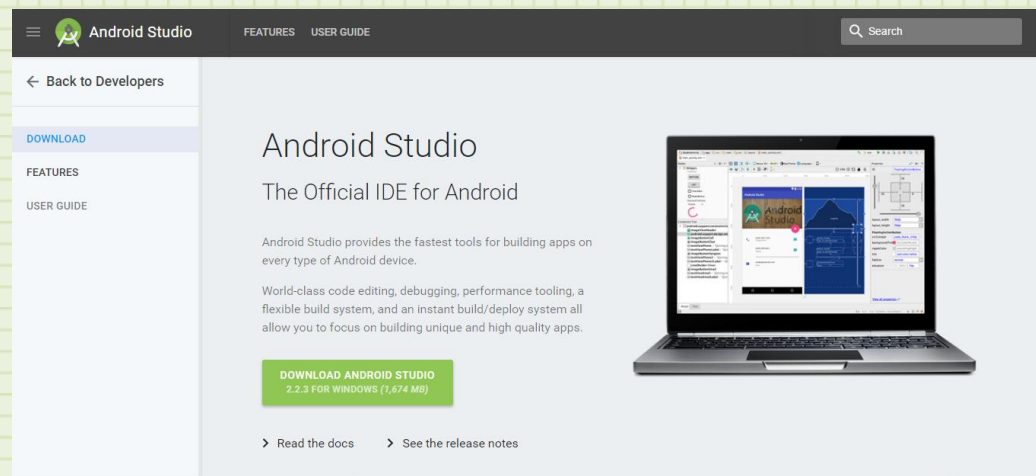
## Section 2: Android Studio

Android Studio is an Integrated Development Environment (IDE) from Google used for writing Android Apps. It has so many features they can't all possibly be covered in an introduction course like this. It can be used to write simple programs like we will program. But, it is powerful enough to tackle complicated games and complex connected productivity applications. And amazingly, it is free to use.

Loading Android Studio on your computer is pretty straight forward. Go to the Android Studio Download page.

<https://developer.android.com/studio/>

You will see something similar to the following:



If the download button doesn't display your particular operating system (OS), scroll down to the bottom of the page and you will see links to alternate OS versions.

**DOWNLOAD ANDROID STUDIO**  
2.2.3 FOR WINDOWS (1,674 MB)

VERSION: 2.2.3.0  
RELEASE DATE: DECEMBER 6, 2016

Select a different platform

Platform	Android Studio package	Size	SHA-1 checksum
Windows (64-bit)	<a href="#">android-studio-bundle-145.3537739-windows.exe</a> Includes Android SDK ( <b>recommended</b> )	1,674 MB (1,756,130,200 bytes)	272105b119adbcbabab114abee4c78f3001bcf7
	<a href="#">android-studio-ide-145.3537739-windows.exe</a> No Android SDK	417 MB (437,514,160 bytes)	b52c0b25c85c252fe55056d40d5b1a40a1ccd03c
	<a href="#">android-studio-ide-145.3537739-windows.zip</a> No Android SDK, no installer	438 MB (460,290,402 bytes)	8c9fe06aac4be3ead5e500f27ac53543edc055e1
Windows (32-bit)	<a href="#">android-studio-ide-145.3537739-windows32.zip</a> No Android SDK, no installer	438 MB (459,499,381 bytes)	59fba5a17a508533b0decde584849b213fa39c65
Mac	<a href="#">android-studio-ide-145.3537739-mac.dmg</a>	434 MB (455,263,302 bytes)	51f282234c3a78b4afc084d8ef43660129332c37
Linux	<a href="#">android-studio-ide-145.3537739-linux.zip</a>	438 MB (459,957,542 bytes)	172c9b01669f2fe46edcc16e466917fac04c9a7f

If you need a version for Mac or Linux, it can be found [here](#).

Click on the applicable download, agree to the terms, and start downloading the installation program.

Google has a good video of the installation process if you require extra help. You will find that on the page you are directed to after you click to download. But the installation is fairly automated so if you have installed anything else before, this should be pretty easy.

You will be presented with the following opening screen:



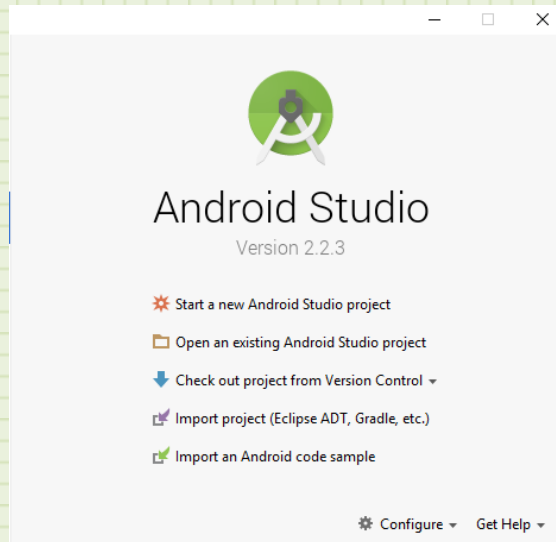
Continue to progress through the installation screens. Click “Next” and “I agree” as needed and keep all of the default selections on each install screen. Once you click “Install,” it will take a while to load all of the required files.

When the installation is complete, start Android Studio. It will take a while for it to start and finish loading. You may see a pop up like the following:



Android Studio looks for updates when it loads so you may be prompted to update. At this point, just hover over the warning and you will see an X show up to close it. Close it. Whenever you want, you can perform updates to the items detected, but it isn’t necessary right now.

You will eventually be brought to an opening screen. This screen will give you some selections to start a project.



Are you ready? Let's start your first project.

## App Idea: What do I weigh on Jupiter?

Let's say you just saw an interesting TV show about space travel and they talked about the difference between the gravity of Earth and Jupiter. It got you thinking: "It would be fun to have an app that calculated my weight on the other planets."

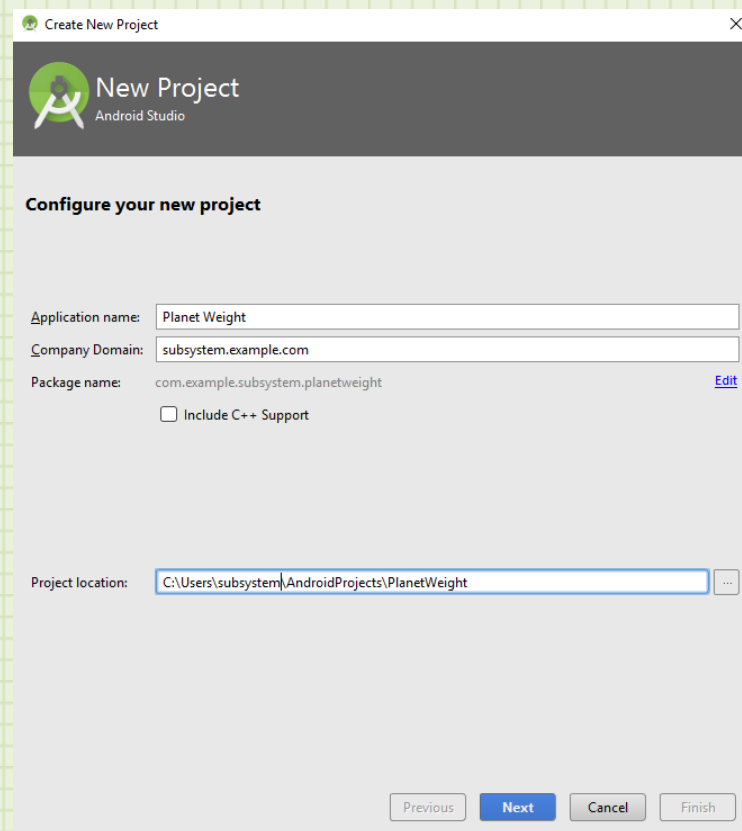
Not satisfied with just thinking about it, you set out to actually create this app.

## Starting a Project

Click **"Start a new Android Studio project."**

Starting a new project initiates an app wizard which will do most of the setup for you. You just need to give it some basic information about the app name and what type of app it will be.

At the first screen, you will be given a chance to name this app. In the block labeled “Application name:” type in “Planet Weight.”



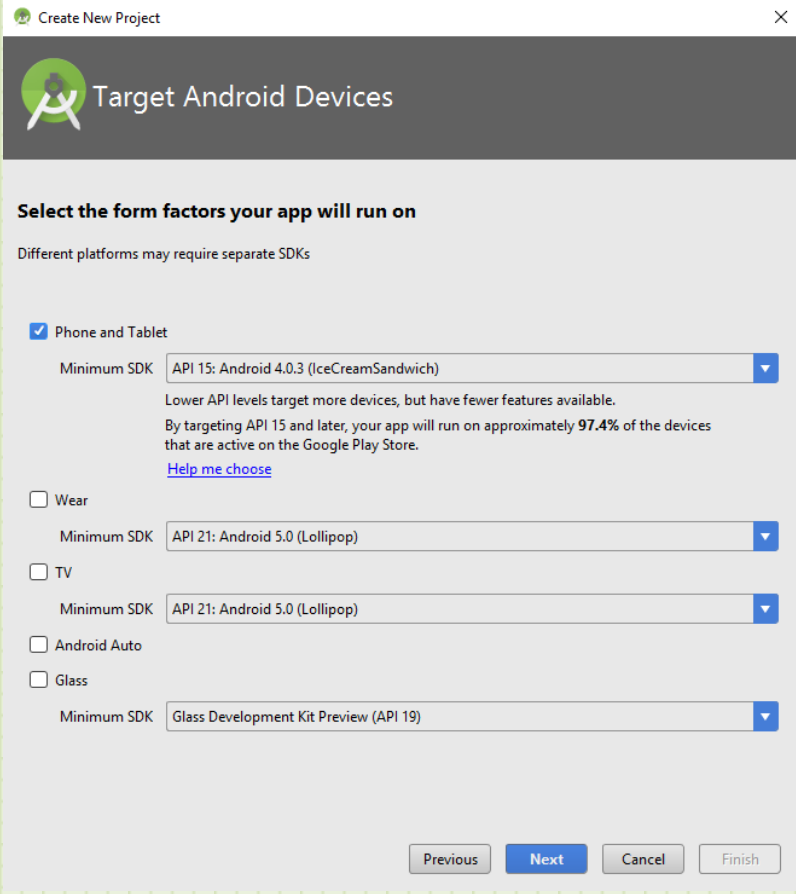
The screenshot shows the 'Create New Project' dialog in Android Studio. The title bar says 'Create New Project'. The main header area has the Android Studio logo and the text 'New Project' and 'Android Studio'. Below this, the section 'Configure your new project' is visible. It contains four input fields: 'Application name' with the value 'Planet Weight', 'Company Domain' with the value 'subsystem.example.com', 'Package name' with the value 'com.example.subsystem.planetweight' and an 'Edit' link, and 'Project location' with the value 'C:\Users\subsystem\AndroidProjects\PlanetWeight'. There is an unchecked checkbox for 'Include C++ Support'. At the bottom, there are four buttons: 'Previous', 'Next' (highlighted in blue), 'Cancel', and 'Finish'.

In the block that says “Company Domain,” you can just leave this as the default value. If you decide to form a company that writes apps, you can change this to match the company name. Right now, it doesn’t have any effect on the project creation.

Click the “Next” button at the bottom.

You will now be presented with a dialog that lets you choose which version of Android to build this app to. There are some decisions to be made here as a app developer. If I choose an old version of Android, I will make my app available to more people. If you think your app may be popular in areas of the world that are not as technologically up to date as other countries, you may want to lower the version to allow older phones to run it. The problem is that exciting features are added to Android all the time and selecting an earlier

version may not allow you to take advantage of these. For now, just use the default setting.



The screenshot shows the 'Create New Project' dialog box in Android Studio, specifically the 'Target Android Devices' screen. The dialog has a title bar with the Android Studio logo and the text 'Create New Project'. Below the title bar is a dark header with the Android Studio logo and the text 'Target Android Devices'. The main content area is titled 'Select the form factors your app will run on' and includes a note: 'Different platforms may require separate SDKs'. There are five form factor options, each with a checkbox and a 'Minimum SDK' dropdown menu. The 'Phone and Tablet' option is selected with a blue checkmark. Its 'Minimum SDK' is set to 'API 15: Android 4.0.3 (IceCreamSandwich)'. Below this dropdown, there is explanatory text: 'Lower API levels target more devices, but have fewer features available. By targeting API 15 and later, your app will run on approximately 97.4% of the devices that are active on the Google Play Store.' and a link 'Help me choose'. The other four options are 'Wear', 'TV', 'Android Auto', and 'Glass', all of which are unselected. Their 'Minimum SDK' dropdowns are set to 'API 21: Android 5.0 (Lollipop)' for 'Wear', 'TV', and 'Android Auto', and 'Glass Development Kit Preview (API 19)' for 'Glass'. At the bottom of the dialog are four buttons: 'Previous', 'Next' (highlighted in blue), 'Cancel', and 'Finish'.

Create New Project

Target Android Devices

Select the form factors your app will run on

Different platforms may require separate SDKs

☒ Phone and Tablet

Minimum SDK API 15: Android 4.0.3 (IceCreamSandwich)

Lower API levels target more devices, but have fewer features available.  
By targeting API 15 and later, your app will run on approximately 97.4% of the devices that are active on the Google Play Store.  
[Help me choose](#)

☐ Wear

Minimum SDK API 21: Android 5.0 (Lollipop)

☐ TV

Minimum SDK API 21: Android 5.0 (Lollipop)

☐ Android Auto

☐ Glass

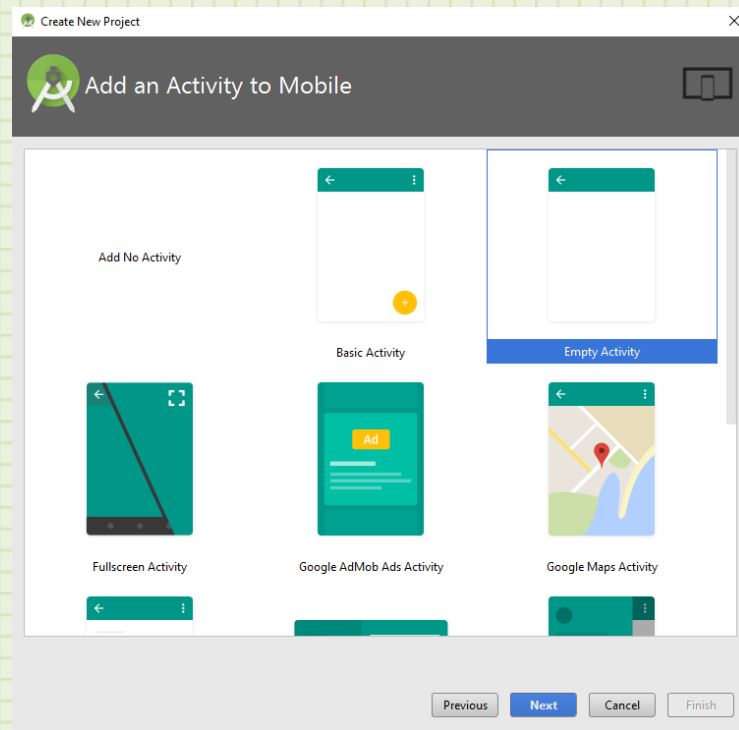
Minimum SDK Glass Development Kit Preview (API 19)

Previous Next Cancel Finish

Press the “**Next**” button at the bottom of the screen.

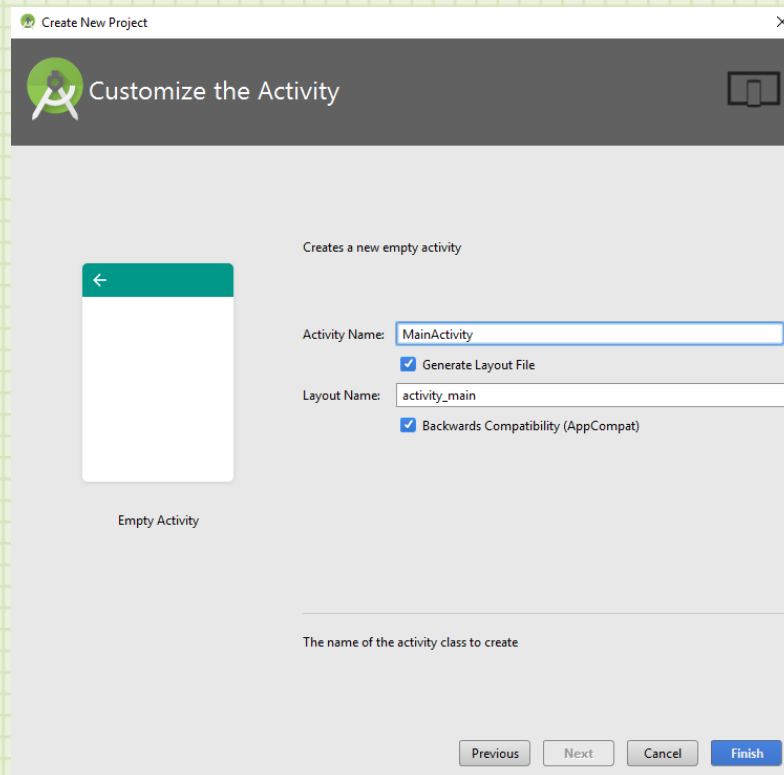
Next, Android Studio presents you with a number of standard app packages. This can speed up your coding considerably because the wizard will automatically add the code and settings required to support these types of apps. Your app may consist of many Activities, but starting out with a prebuilt one for the task you wish to do makes customizing it and running it much quicker. It is also a great chance when you are starting to learn the basics of creating an app with the wizard and seeing how Android Studio sets up the different kind of app structures.

We are going to start from scratch, so select the “**Empty Activity**” and then click the “**Next**” button.



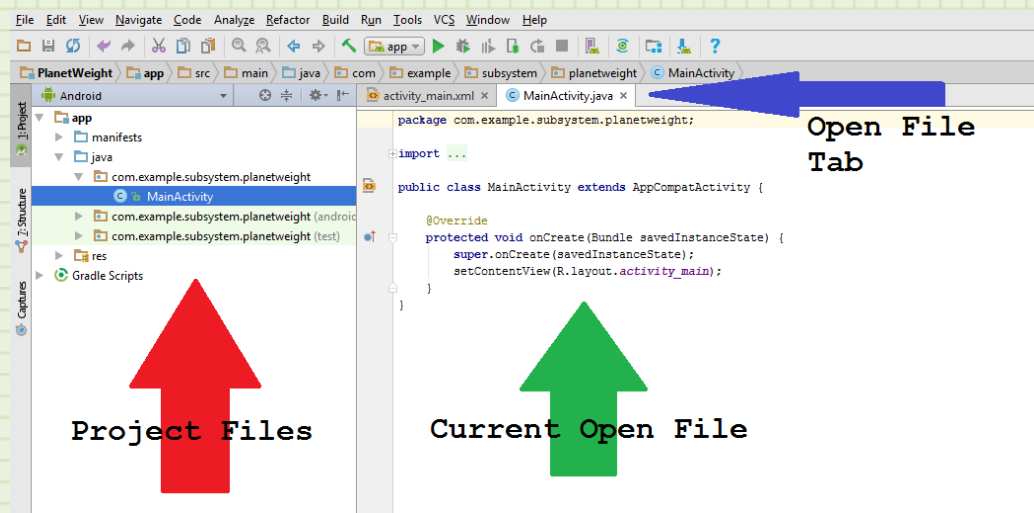
Now, Android Studio will present you with the ability to change the Activity and Layout names if you desire. This is not the same as the App name that you set at the opening screen. This will just be the name of the Activity (remember we talked about an Activity as something the user can interact with – like a screen with a button). An app can have many Activities so it is just asking if you want to change the name of the Activity it is about to create.





Leave the Activity and Layout names with the default value.

Click the “Finish” button at the bottom of the page. The wizard will now begin the long process of establishing and configuring your new project. You can watch the status in the lower status block. Give it time. Depending on your computer, this can take several minutes. When it is complete, you will see a screen similar to the one below that is highlighted with important areas.



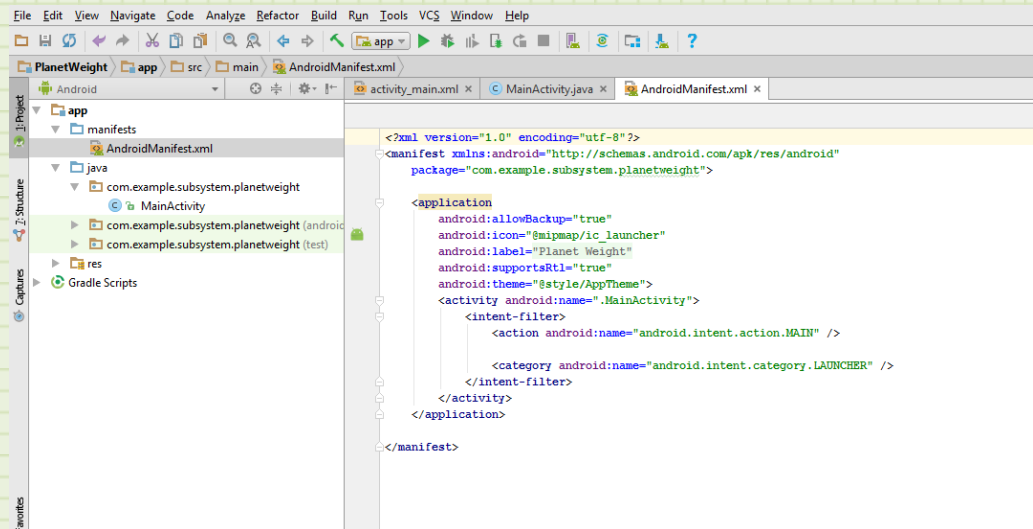
The section highlighted by the **red arrow** is where you will find the files associated with your project. Here you can open Java, Layout, Resource, and Configuration files.

The **green arrow** points to the area that displays the currently open file. This is where you will type in data into your project.

The **blue arrow** shows you tabs of all the open files. You can navigate to the different tabs by just clicking them. When you do, you will see the content displayed in the green arrow area.

Android Studio is what is known as an Integrated Development Environment (IDE). These are very convenient and powerful systems to allow you to write code, compile applications, run them on simulators, debug (or fix) problems, interact with the app while running, and combine other prebuilt code into your own application. They make the job of coding much easier and more fun.

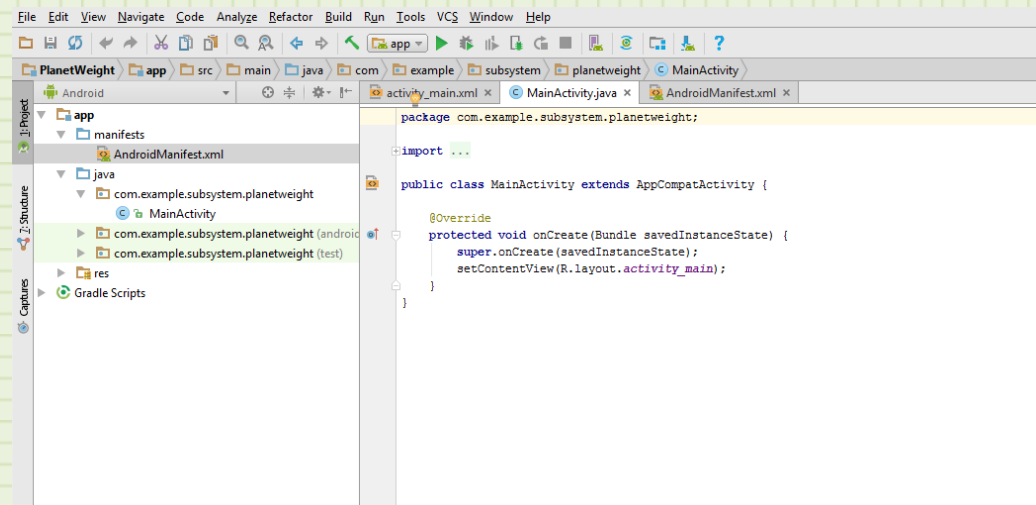
In the navigation tab area, you will see 2 tabs for files that have been opened for us. These are the Layout and Java file created by the wizard. That is two of the three files we discussed earlier. The missing one is the manifest file. If you look at the Project Files section, you will see a folder called **“manifests.”** Click on the gray arrow to the left of the folder to expand what is in it. You will see a file called **“AndroidManifest.xml.”** Double-click this file to open it.



You will see this document open up in the document window. Don't worry that a lot of what you see looks crazy. It should at this point. One thing that you can see though is that there are some very readable things in this manifest. First, the package name appears at the top of the file. This is the package name we selected (or rather, the wizard selected for us at the beginning). Then, there is a section called "**<application.>**" This section will include all of the important information the Android system needs to know about our app. In the first line, we tell the operating system to allow someone to back this app up on their phone. This give Android permission to transfer the app from the phone memory to a memory card in the phone. The next line gives the name of the icon we want to use. It is set to the default right now, but there is an easy way already built into Android Studio to make your own icons so we will definitely do that later. The next line is our app's name.

A few lines down, you will see a section called "**<activity.>**" Each activity in our app must be listed in this manifest or else Android will not allow it to run. This is a safety feature of Android to prevent external activities from gaining access to your app. Early in programming, you may forget this important fact. If you add another activity to your project, like a simple pop-up selection dialog, and you do not add it to the manifest, your app will crash every time you go to run it and you will get frustrated looking for the problem in your code when the problem is in your setup. Keep that in mind if your app keeps crashing and you don't see any obvious problems.

In the tab navigation section click on the tab labeled “MainActivity.java.” This will bring up the contents of this file on the open file section.



You can see right now, there is not a lot going on in this file just yet. When we selected Blank Activity in the wizard, you might expect that we would get a blank activity. And yet, there is a little bit of code here. The code really is the absolute minimum needed to create the activity. This file consists of the package name that this file is attached to, a line that defines this activity and associates it with an already well established activity type:

```
public class MainActivity extends AppCompatActivity {
```

There is a bunch going on here. But what this line says is to create a **class** called **MainActivity** that is an **extension** of the basic class **AppCompatActivity**, and make that resulting class **public** so other code can have access to it.

A **class** is a piece of code that creates an object that we can later use, modify, and discard. It is used as the primary means of programming in object-oriented programming languages like Java, C++, and Python. Classes allow us to take objects that have already been created and modify them for our use with little additional code. For instance, we will add a button to our app to calculate the Planetary Weight of a person. We will not write any of the code for the functionality of that button. We will use a Button class that already knows how to draw itself, operate itself, and report if it has been clicked. We will just tell our code the button is there and instruct it to respond to the

clicking. These classes have made programming much easier and allowed the reuse of classes in other programs. This is good news for programmers. If you spend days getting the look and feel of a particular user interface component correct, you can make it into a class and use it across many different operating systems.

The `Public` term above is what is known as an access identifier. It determines what parts of your code have access to this activity. There are three primary access identifiers:

1. **Private:** This restricts the access to the class itself. Only code that is a part of the same class can access code in the class.
2. **Protected:** This allows the class itself and all its subclasses to access the class.
3. **Public:** This means that any code can access the class.

These different access levels help programmers keep their code protected. This allows them to select names for functions and variables that might be common to not be accessed by other code that may use those names in their own code segments. This allows you to have a variable named “color” in your code section, and it doesn’t interfere with the variable “color” in the main program. Just set the variable “color” as private in your code. Then, code outside that section can’t even see it.

Again, don’t worry if this is a little confusing. You will not need to understand it all perfectly to write your first program. If these concepts are introduced a little at a time, it will help de-mystify some of the words in your code and make you feel a little more familiar with why they are there.

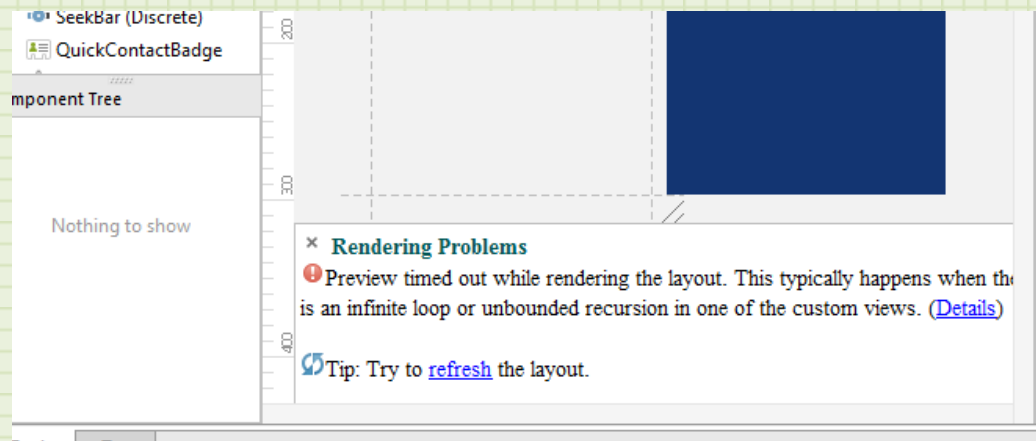
A few lines down, you will see the declaration of the `onCreate()` function. This should look familiar. This was one of the functions in the Activity Lifecycle discussed earlier. If you remember, this function is called when an activity is first started.

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

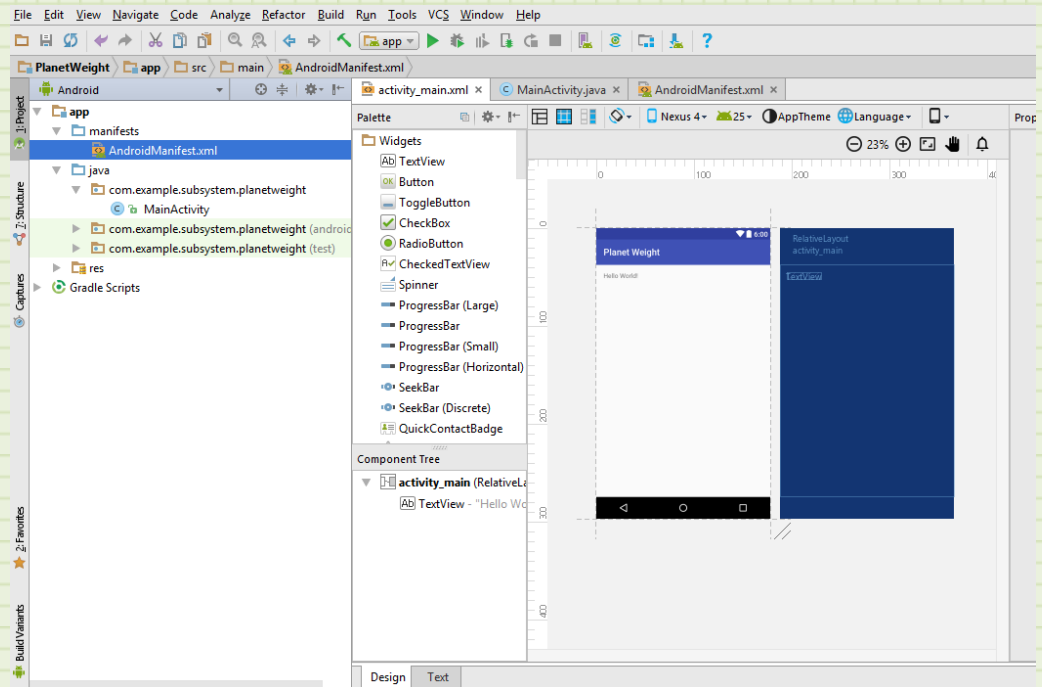
Notice the setContentView() function. It has the name of the layout file the wizard created for us. This is the place in our code where that layout is loaded to show the user.

Let's look at that layout file. On the open file navigation tab, select the **"activity\_main.xml"** tab.

This opens the activity layout file. Here you will see the graphic picture of how your activity is going to look.



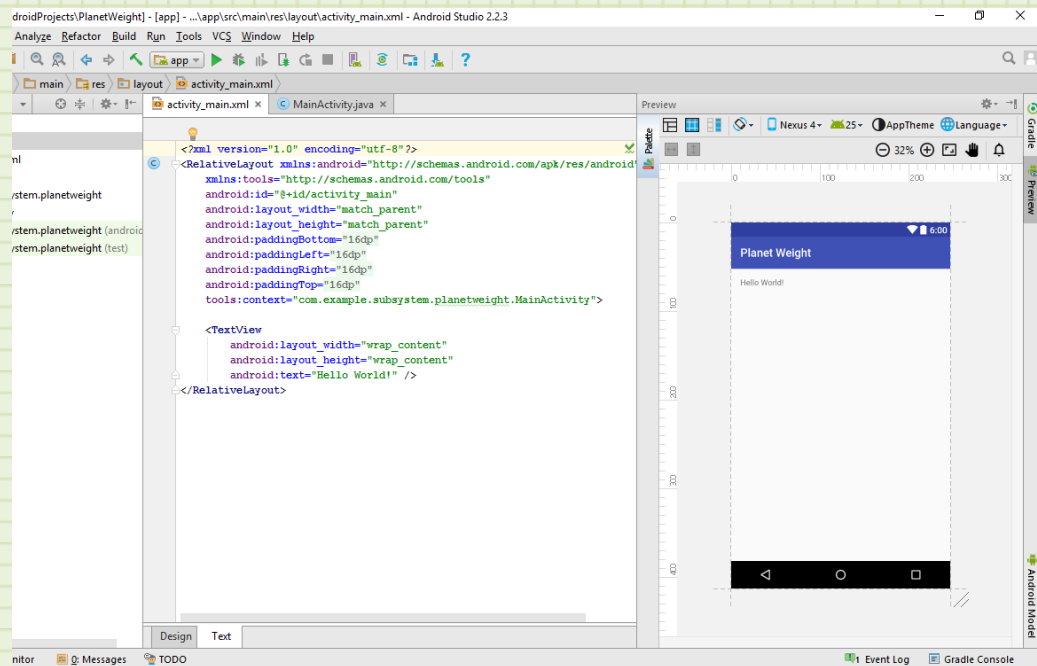
Sometimes, your project will take a long time to load. You may receive a warning in your layout file. If this happens, click "refresh" on the warning and this will usually clear the problems.



In this view you are given a preview of how your interface is going to look. This area allows you to drop in new components, rearrange existing components, and set the values of many items. Right now, the layout only consists of a title bar with the app name in it (Planet Weight), and a text box that says the words “Hello World.” If you have ever learned another programming language, you might be familiar with this. The first lesson of most programming languages is simply to display “Hello World” on the screen.

On the bottom of this section are two tabs. You are currently viewing the “**Design**” tab. This gives you how your UI will look. The “**Text**” tab is what the actual layout file looks like.

Click on the “**Text**” tab.



Here, you can edit the actual text of the layout file. It also shows you a preview of the UI so you can see the effects your changes have. Right now, you can see our layout has an element called “<RelativeLayout.” This is a container that we can put components in and draw them relative to the borders, any margins we establish, and other components. This really gives us a drag and drop interface. We can select objects from the components menu, drag them to the UI, and drop them in the position we want them. Android Studio then generates the correct layout code to make this happen. This is convenient. But sometimes, dragging doesn’t quite get you the results you need and you have to go to the “Text” tab and enter the desired values manually. Android Studio gives you both options.

There is currently only one component in this **RelativeLayout**. It is a **TextView**. It is a simple box that displays text. Currently, it is loaded with the text “Hello World.”

It may not feel like it since we haven’t written any code, but we have a complete Android app right now. The wizard has done its job and setup everything this app needs to load itself in the Android operating system, register its components, and initiate an activity lifecycle with our newly



created activity. In the next section we will setup a simulator and run this app on an actual simulation of an Android device.

## **Section 3: Simulate and Run**

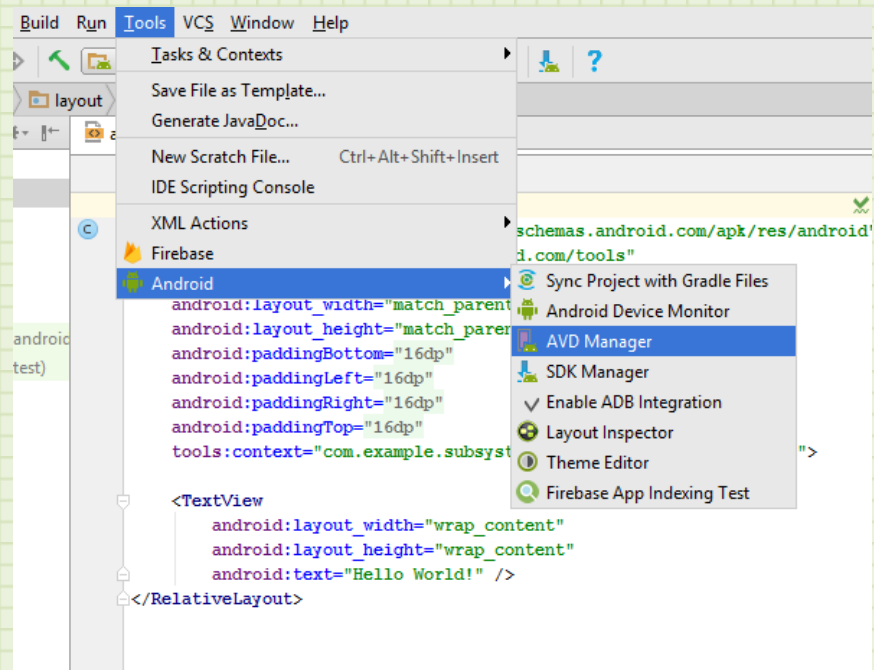
Android Studio is a powerful app writing platform. On top of allowing us to write the code for the app, visually create the UI, and highlight errors and suggestions, it also allows us to run our apps on simulated hardware. In this section we will create a virtual phone, compile our app, and run the app on the created phone. And we will do all of this from within Android Studio.

### **Create a Virtual Device**

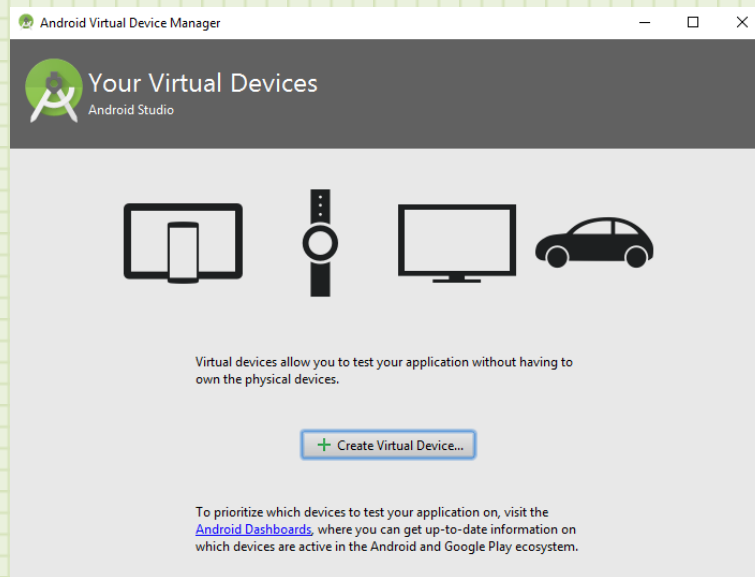
Android is an operating system that runs on phones, tablets, watches, TVs, and other smart devices. Android Studio allows us to create a simulated device so we can test how our apps will look on these different platforms. For our app, we will just look at a smart phone.

To set up a virtual device, you will use the Android Virtual Device Manager. This is a single location that you use to create, modify, store, and launch all of your virtual devices.

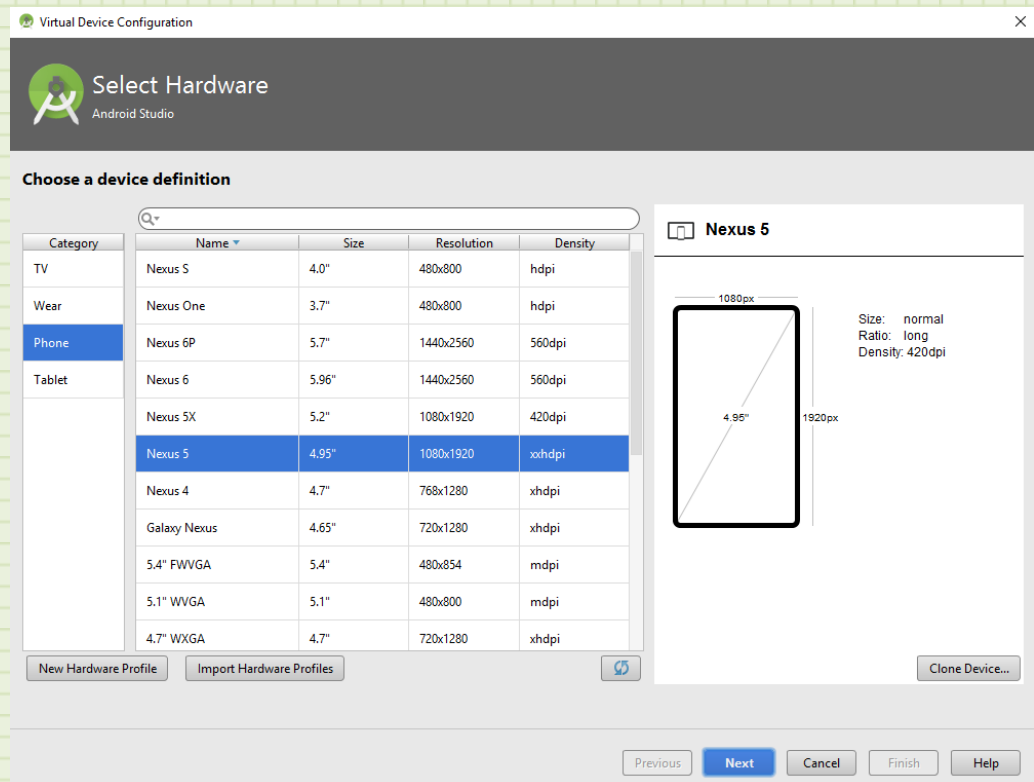
Select **Tools->Android->AVD Manager**



When you select this menu item, the AVD Manager will open a dialog like the following.



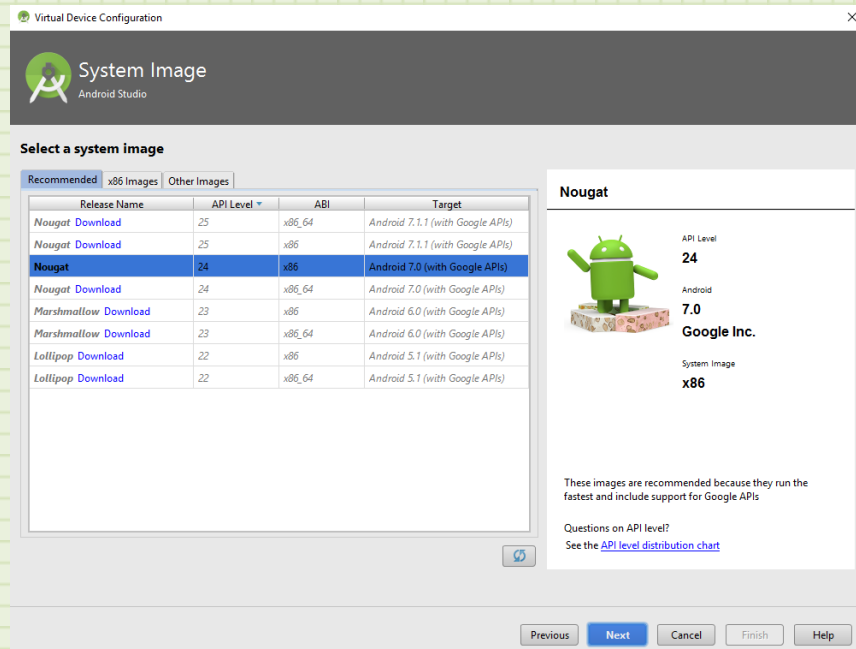
Select the “Create Virtual Device” button. This brings up the Manager.



The Manager will give you option at this point to create a virtual device. On the left, you will see a Category window that lets you select which type of device you want to test on. Select the **“Phone”** category if it is not already selected. There are a lot of prebuilt options for devices, so we will take advantage of this. Let’s build a Nexus 5 phone to test our app. Select **“Nexus 5”** as the phone if it is not already selected. The window on the right gives you some of the basics of what user interface it uses.

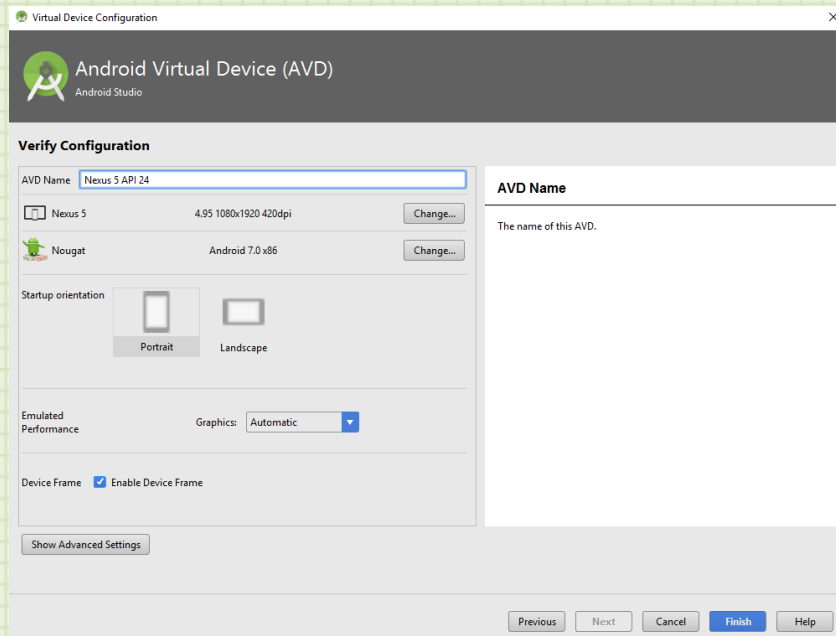
Click **“Next”** at the bottom of the dialog.

The next section of the setup wizard allows you to specify which level of Android is run on the device. These different options may come in handy later in your development. But right now, we just want a device to run our app so we will stay with the default value.



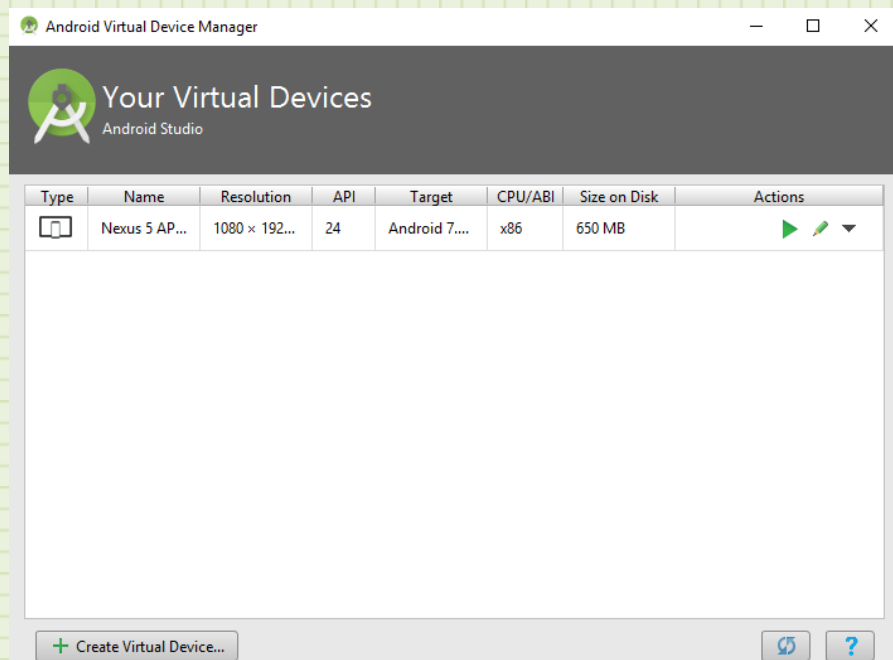
Click **“Next”** at the bottom of the page.

Finally, you come to the last page of the wizard. Here, you can set some of the behaviors of the phone. You can give the phone a special name. This is important if you have multiple version of the same type of phone. You could then name them with their specifics. Again, the name is fine so we will just stick to all the default values.



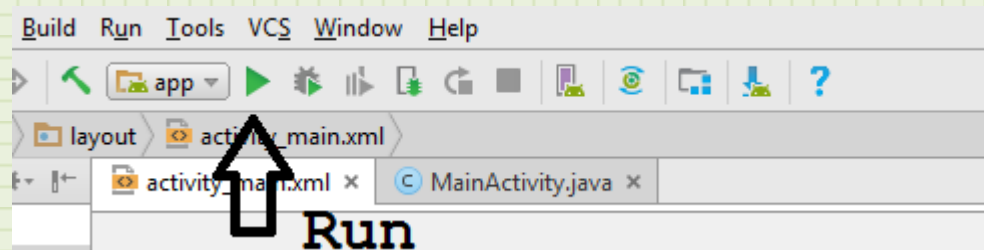
Click **“Finish”** at the bottom of the page to generate your test phone.

When the software is done creating the device you will be brought to the device summary page. Here you will see all the devices you have created, the status of each, and you can start and stop them from here.



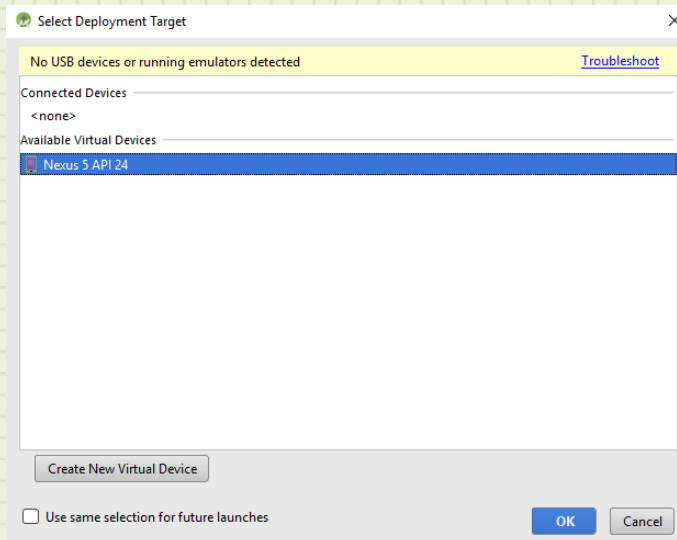
You can see that we just have the one entry. This is the Nexus 5 simulator you just created. Close the AVD Manager by clicking the “x” in the upper right corner of the window. This will return you back to the main Android Studio program.

Now we are ready to run our program. On the toolbar, there is a little green triangle. This is a button to run the current project.



Press the “Run” green triangle to run the program.

You will be greeted with a dialog that looks like the following.

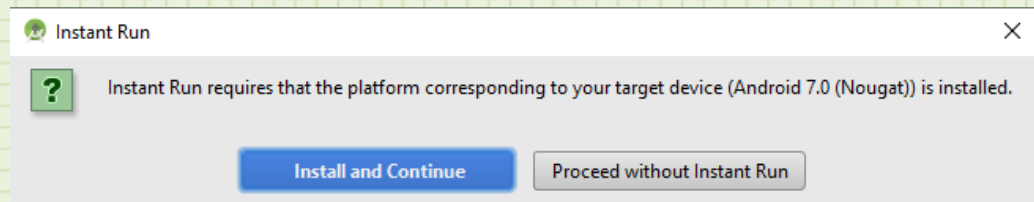


Here, Android Studio is asking what you want to run the compiled program on. You will most likely see the only option we have is the simulator we just created. Select it and click “OK” at the bottom of the page.

This is going to do two things.

- 1) Android Studio will start compiling the program
- 2) The simulation of the phone will start to load

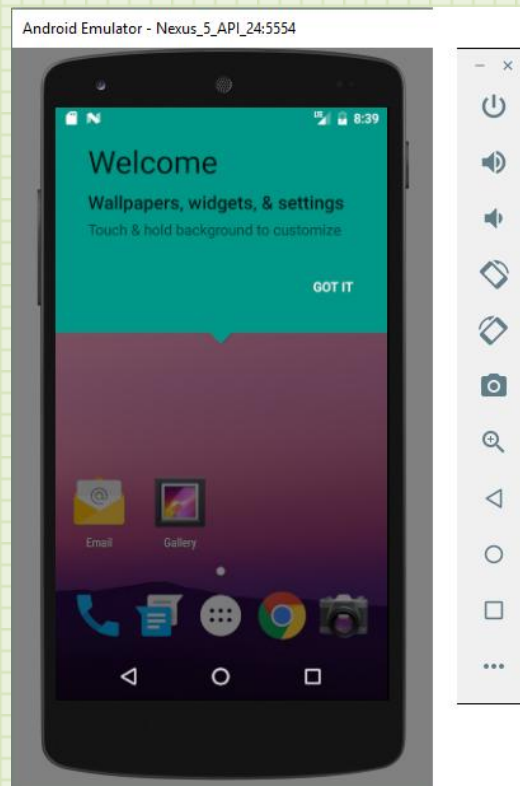
You may receive the following message. Just click “**Proceed without Instant Run**” for now.



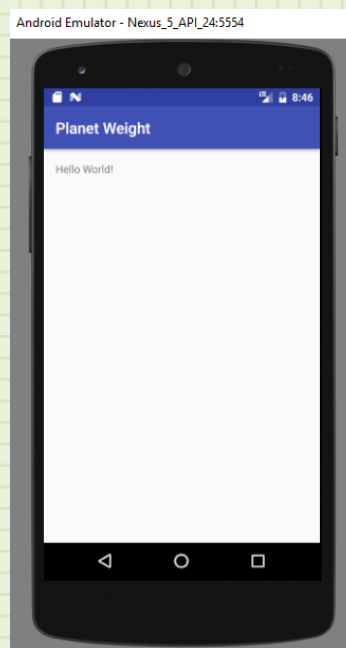
This will probably take a long time. Be patient. You can watch the progress in the lower display bar in Android Studio. The messages probably won't make a whole lot of sense, but you can see the progress and any messages in the lower windows. What Android Studio is doing is compiling your program into an app. This requires a lot of processor power to convert, merge, copy, and all the other operation it takes to bind all of the information together. The simulator will most likely take a long time to load (could be several minutes). The simulator uses a lot of computer resources so minimize anything else you are doing on the computer while the simulator loads. Once it is running, Android Studio will send the compiled application to the phone and launch it on the screen.

At some point, you will see a window pop up with the image of an Android phone on it and a menu off to the right. This is the phone emulator. It will operate just like an Android phone will. Later, if you are not familiar with Android phones, you can play with the different buttons, turn it off and on, and run different apps that are preinstalled. For right now, just wait until the process is done and our app is displayed.





If all went well, you will eventually see the phone with your app loaded on it.



Ok, there is not a lot to it. But the fact remains; it is a full Android app with the ability to load, unload, receive commands from the operating system, place itself in different modes of operation, and display our text on the screen. This is a very important step in your app development. If we couldn't run our apps, we wouldn't know if they worked or not. Feasibly, you could develop apps for Android on a MacOS machine, run it on the simulators to show they are working, and post them to the Android App Store without ever testing them on a real device and have a reasonable belief that they will function correctly. Pretty amazing.

This is a very big step. From here on, we will be modifying our app to make it continually better and running it at each step of development. If you have run this simple app, you know that the basic framework is correct. If you get an error from here going forward, you will know that it is code you added or modified that is causing the problem.

Let's get back to our app and add some real functionality.

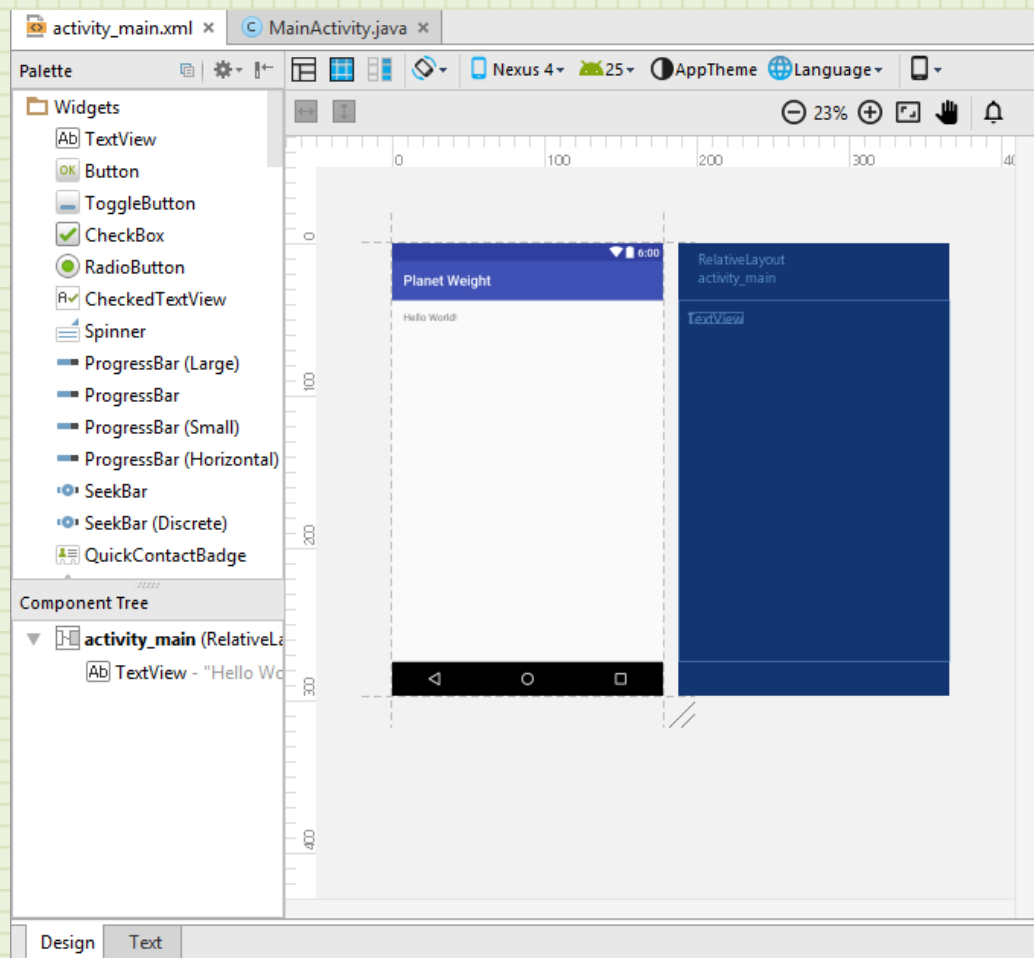
## **Section 4: Adding Functionality**

At this point, we have a working app that doesn't do much. Let's change that.

It is important to mention at this point that the intent of this set of lessons is not to make you an expert programmer but to teach you how to get up and running quickly. There are many ways to learn programming. This approach is meant to get you programming, show you how things interact in Android Studio, and give you enough information to program a simple app. From there, if you enjoy programming, we encourage you to look at the resources in the appendix to learn more. So if this starts to look complicated, don't worry. Programming is a thing that you can constantly grow and get better at. Over time you will see that many of your skills will be transferrable to many other programs you want to write and many things that confused you will become easier.

### **Create the User Interface**

Open project we started in Android Studio. On the tab navigation or from the project navigation section, select the file `activity_main.xml`. This is the layout file that had our "Hello World" TextView in it. We are going to add some components (referred to as widgets) to give our app the interface it needs.

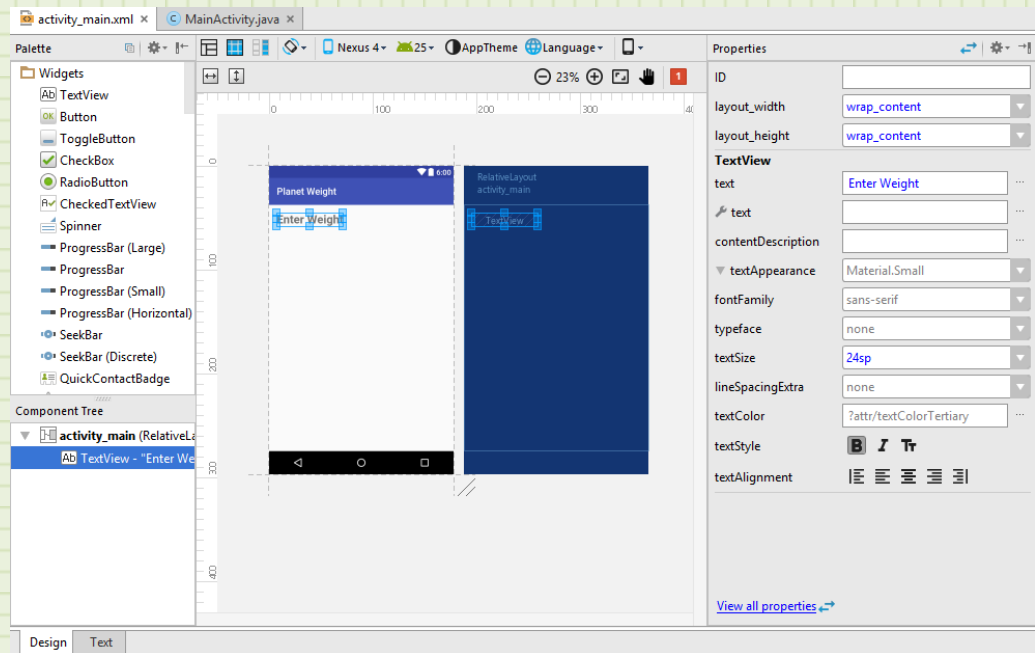


Based on the overall idea of our app, we will need the following widgets in our UI.

- 1) A **TextView** that states “Enter Weight” so the user knows what goes in the input box. (We will just re-purpose the “Hello World” widget)
- 2) An **EditText** that allows the user to enter numbers for the weight.
- 3) A **Button** that calculates the weights on the different planets when clicked.
- 4) A **TextView** that displays the output of our calculation to the user.

One of the easiest ways to program apps is to lay out all of the UI elements first, and then write the code that makes them all interact. So we will go ahead and build all of these elements into our UI.

On the left side of the layout file you will see a window with available widgets. Below that, there is a navigation pane that has all of our current widgets in it.

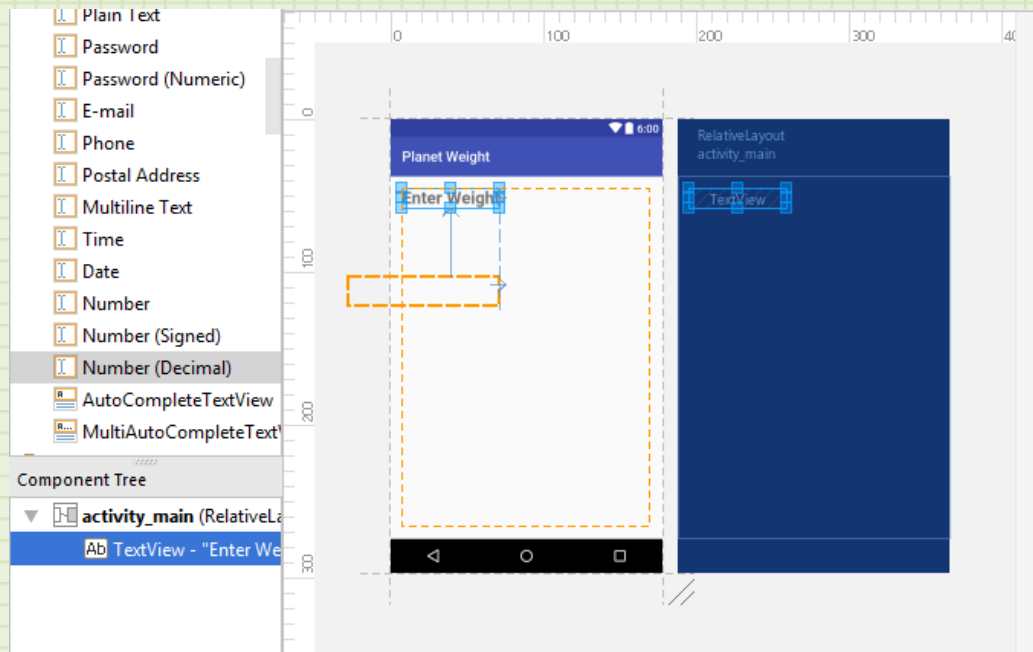


Click on the TextView in the Component Tree to make our TextView active. You will see a Properties window appear on the right. This is a convenient place to change some of the attributes of a widget. In the box marked “text” on the right, click it and enter “Enter Weight” into that field. Down where it says “textSize,” select “24sp” to make the text larger. Also select the **bold** icon where it says **textStyle**. This completes the look of our prompt. You can see the changes we have made on the graphic display in the middle of the dialog. The text now says “Enter Weight” and the text is larger and bold. You are starting to see the benefit of object-oriented programming. This TextView was a widget that was already created. We can use it and customize it without having to write any code. We just changed some of the available properties and we have the TextView we want. This is a lot easier and quicker than having to write all the code to make these things happen.

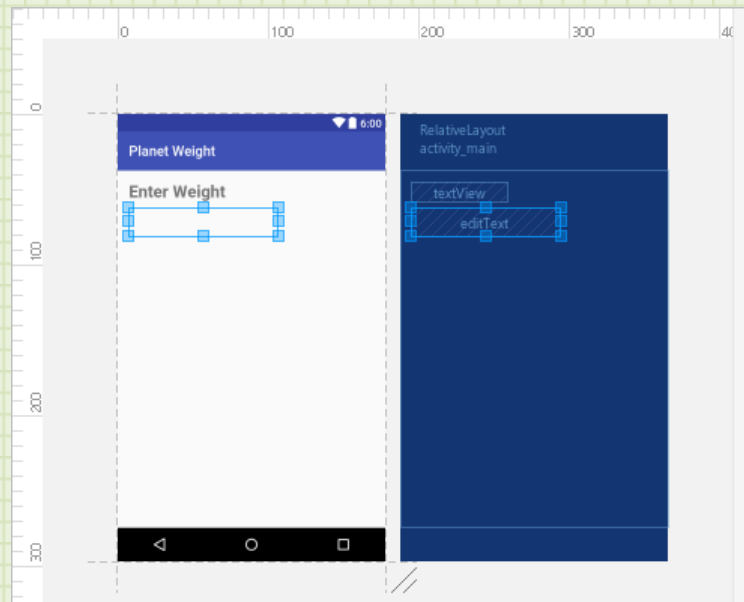
Now let’s add the input box.

We want the user to be able to input their own weight into the program so we need a widget called EditText. This component handles input from the user

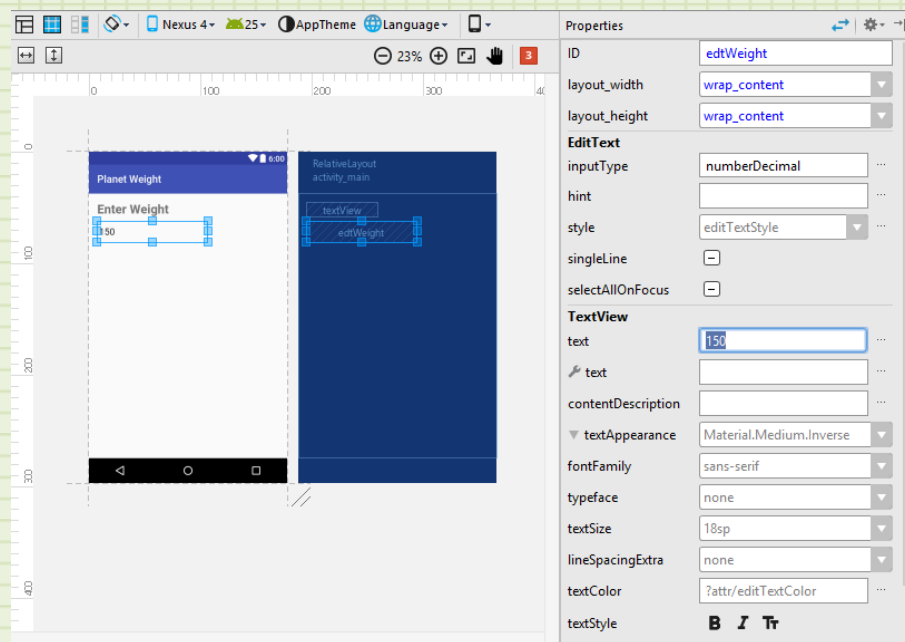
and allows us to retrieve the input in our programs. From the widget menu, click and hold to drag a Number (Decimal) widget. As you drag widgets onto your UI the software will give you guides to show you where you are relative to the borders of the screen and other widgets. When we looked at the text in the file `activity_main.xml` there was a widget called `RelativeLayout`. That is what you are dragging widgets onto. It will handle writing the code for the placement of widgets. Drop the `EditText` on the screen so that it is under the `TextView` and flush with the left side of the screen.



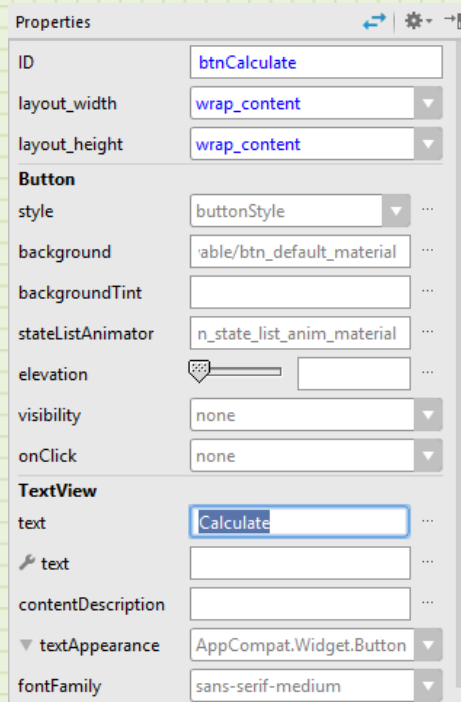
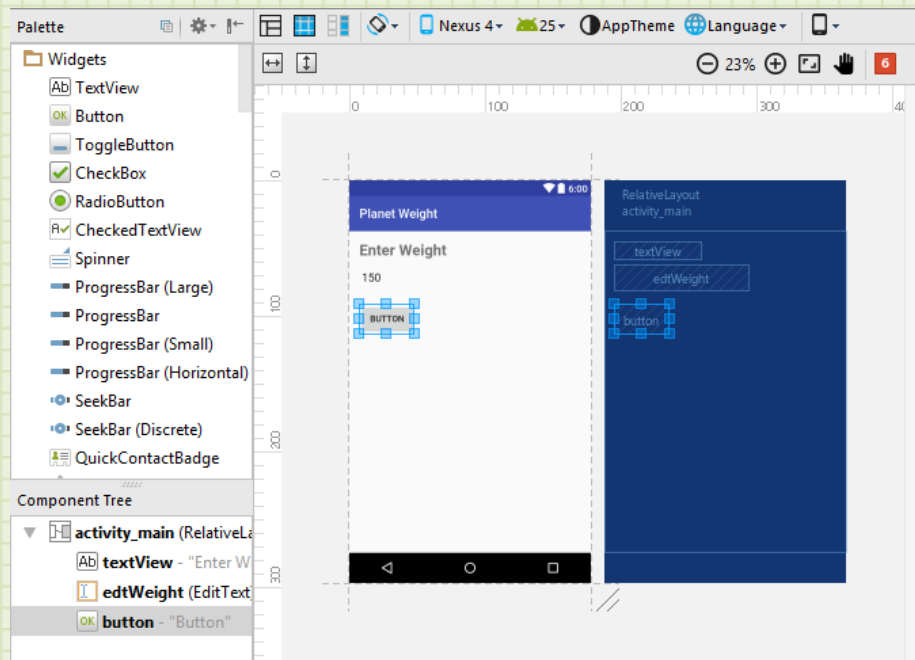
When you are done, your widget should be positioned similar to the one shown below. If something crazy happened and the widget wound up someplace you don't want it, you can always grab it again and replace it where you desire.



Now let's set up the EditText like we did the TextView. In the Properties section, change the **ID** of the EditText to **edtWeight**. Also, change the default **text** to **"150."** This will give a default value of the weight when the app is run.



Now add the Button. From the widget menu, grab a Button and place it below the EditText and flush with the left side of the screen.



When you are done, it should look similar to the above layout.

Now, edit the Properties of this Button widget.

Change the **ID** to **btnCalculate**, and the **text** to **“Calculate.”** You can leave the other properties with the default values.

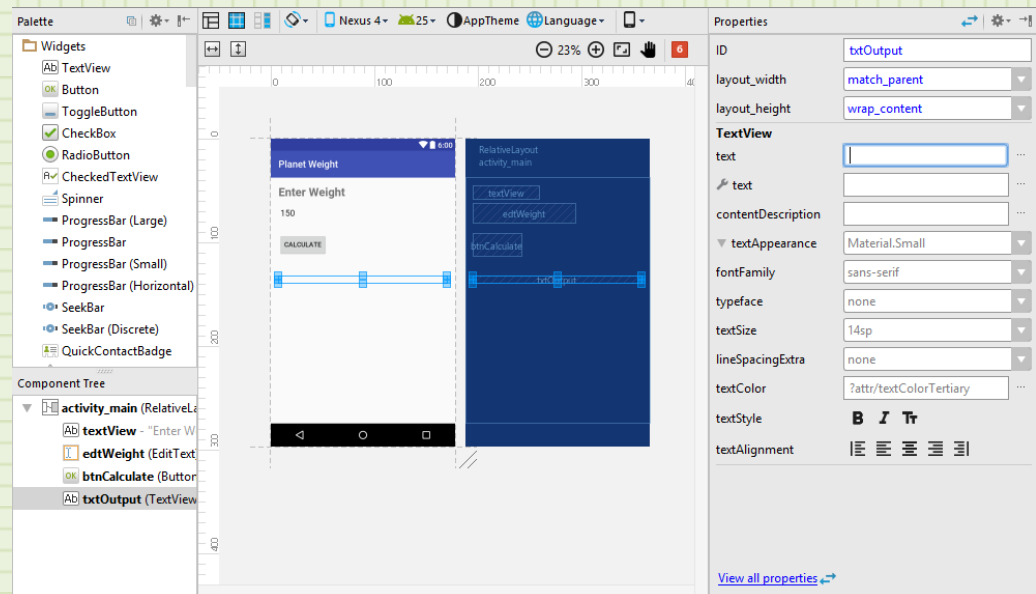
Again, you can see these changes in the graphical screen to ensure you are making the changes you desire.

Most widgets have the properties **layout\_width** and **layout\_height**. These define how the widget will display on the screen. If you select **wrap\_content**, the

widget will size itself to fit the content of the widget. If it is set to **match\_parent**, it will expand to fill the parent widget it is in. We will use that in our last widget that we will add.



Add a **TextView** widget from the Palette to the screen so it is placed below the Button we just created and is flush to the left.



In the Properties, change the **ID** to **txtOutput** and the **layout\_width** to **match\_parent**. Notice how the TextView now spans across the entire screen.

This completes all of the UI elements we said we needed. Save your work by selecting **File->Save All** from the menu or by clicking the Save icon on the menu bar.

Let's run our app to see what this looks like and what capabilities these widgets have already added to our app.

Click the Run icon on the menu bar (the green triangle).

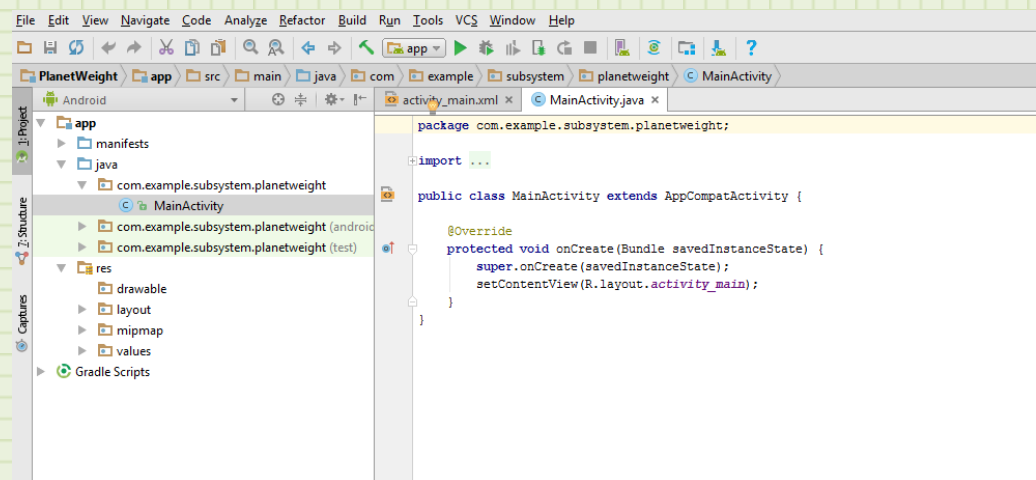
As before, you will be asked what to run this app on. Select the emulator we created before (the Nexus 5) and run the app.

After a while of compiling and loading the emulator, you should see your app displayed. Play around with the widgets. Notice that you can press the button but it doesn't do anything. When you click on the EditText box, an online keyboard comes up where you can enter a number. Notice, since we selected a Number (signed) as the type of EditText, the number keypad comes up to

edit instead of the entire keyboard. You can also change the number in the EditText. This is a lot of functionality without typing a single line of code. But we want our app to do something. Close the emulator and switch back to Android Studio. We will now start writing code to make our app work.

## Write Code to Make Things Work

We finished our UI. Now it's time to write some code. In Android Studio, on the file navigation tab select the **MainActivity.java** file.



The first thing we need to do is get our code to recognize the widgets we just placed in our UI. We do this by establishing names that represent these components. The choosing of names during program is important because it helps us know what we are dealing with when we program. If we called our button "A" and our TextView "B," later in the program we might forget which is which and spend time finding where we assigned the name to know which was which. Instead, it is good to come up with naming conventions so that you will recognize the widget by its name.

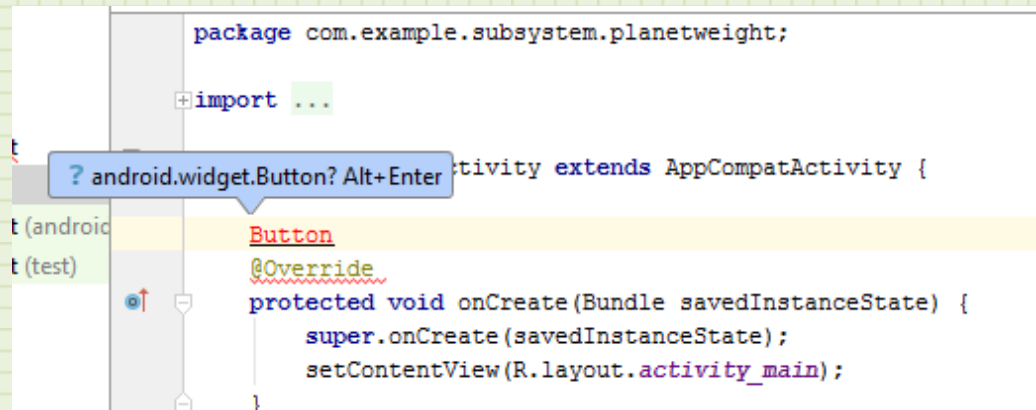
In Java, when we create a Variable Name to represent our Button, it has to be the same type as our Button. So we will define the name of our Button as btnCalc. It is defined with the following line of code:

```
Button btnCalc;
```

This line says “We want to create a variable of type Button and we want to give it the name **btnCalc**.”

This command reserves a place in memory for our button. We haven’t assigned our button to this variable yet, but we are getting to it.

Add that code to the MainActivity.java file right after the MainActivity declaration. You can see this location in the figure below.

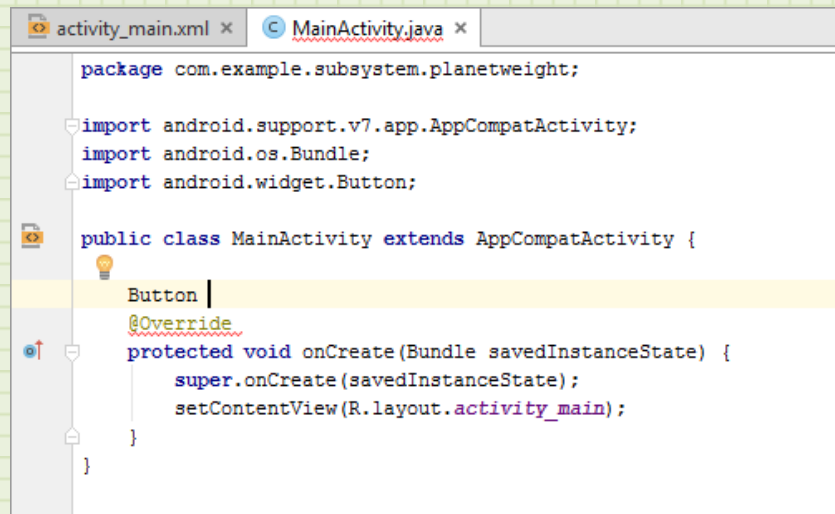


Here, we are going to witness some of the power of Android Studio and why Integrated Development Environments (IDE) are so convenient.

Type in the text “Button.” You will see that the text will turn red. This is the IDE telling you there is a problem. Right now, our project doesn’t know anything about what a Button is. We need to import a definition file. You can see in the figure above that the IDE is recommending a fix to the problem. The blue popup box with the question mark is showing a probable solution. If we hit the Alt key on our keyboard and the Enter key at the same time, the IDE will load the definition file for us and clear the error. If you don’t see the blue box, click on the word “Button” and you will see an options box that will let you load this same definition file.

Where is that definition file?

If you look a little higher in the code area, you will see a section called “import...” This area imports pre-built code that we use in creating objects like the Button. If you click the plus sign next to the “import...” you will expand that section to see some of the things we are importing. Of note, the latest import on the bottom is our Button widget.

A screenshot of an IDE window showing the MainActivity.java file. The code includes package and import statements for AppCompatActivity, Bundle, and Button. The MainActivity class extends AppCompatActivity and overrides the onCreate method. A lightbulb icon indicates an autocomplete suggestion for 'Button' at the end of the line 'Button |'.

```
package com.example.subsystem.planetweight;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.Button;

public class MainActivity extends AppCompatActivity {

    Button |

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

That import has all of the code our button uses to run and allows us to define things as Buttons. Finish typing in the name of the button so the final line should look like:

**Button btnCalc;**

All of our command lines in Java end in a semi-colon.

Let's add the other two variables we need in our program.

Below the Button definition add the other two widgets, and do the same procedure as above to add the definition files. When you are done, it should look like the following:

```

package com.example.subsystem.planetweight;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {

    Button btnCalc;
    TextView txtOutput;
    EditText edtWeight;

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

```

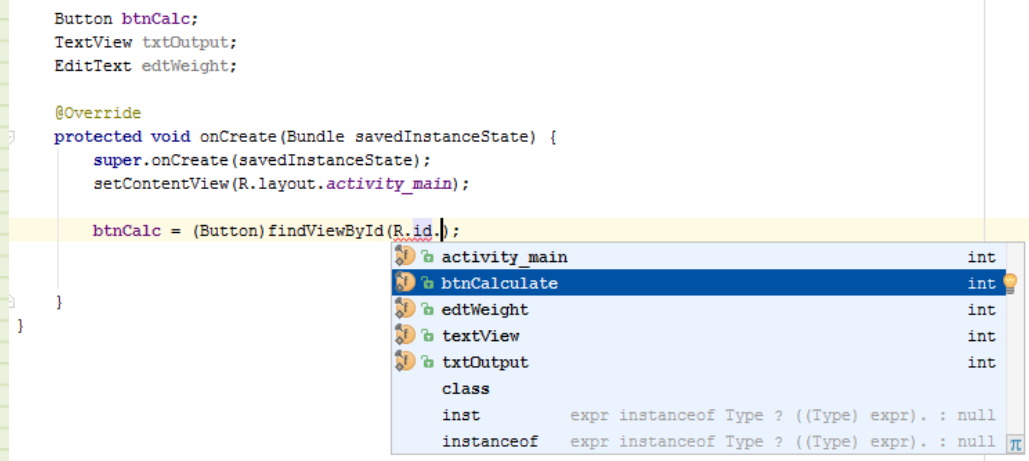
Note how the variable names we just added are light gray. The IDE will color things light gray if they are not being used. Since we haven't done anything with these, they are gray. What we need to do now is assign them to our widgets. Let's start with the Button.

Go down to the section of code after the onCreate function. This is the section of code that is run when the activity is created. So this is where we want to assign our widgets to their variables.

To do this, we are going to use a function that looks up the ID of our button. The function will look like the following:

```
btnCalc = (Button)findViewById(R.id.btnCalculate);
```

What is happening is we are assigning our Button variable "btnCalc" to the actual Button we dropped in the UI. We find this button by looking it up with the findViewById function. As you start typing this line in your code, you will notice that Android Studio will anticipate what you want based on the context of where you are in code. It can speed up entering programming dramatically and also prevent you from making spelling or capitalization errors. As you type the above line in, when you get to the R.id... section you will see the following:



As you type, helpful boxes will show up with these suggestions for automatically completing your typing. In this case, when you type “R.id.” you will then see the IDs of the components in your app. In this case we want to select the **btnCalculate** because that was the ID we gave the button we placed in our UI. Whenever the IDE suggests something, you can scroll down the list and click the selection or hit the enter key and the text will automatically be filled in for you.

Type in the other variable assignments:

```
txtOutput = (TextView)findViewById(R.id.txtOutput);
```

```
edtWeight = (EditText)findViewById(R.id.edtWeight);
```

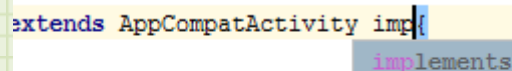
When you are done, the whole entry should look like the following:

```
public class MainActivity extends AppCompatActivity {  
  
    Button btnCalc;  
    TextView txtOutput;  
    EditText edtWeight;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
  
        btnCalc = (Button)findViewById(R.id.btnCalculate);  
        txtOutput = (TextView)findViewById(R.id.txtOutput);  
        edtWeight = (EditText)findViewById(R.id.edtWeight);  
    }  
}
```

Now that we have declared variable names and attached our actual widgets to them, we can interact with our widgets through these variables. We can set properties like color, text, and size by writing code to manipulate them. If you place a widget and are not going to change anything about it (like the text box “Enter Weight” on our UI) then you can just set these when you design the widget. If you need to interact with them in code (like our Button, EditText, and output TextView) you need to assign variables to them so you can have access to their properties during run time.

When we ran our program, we saw that the EditText had all of the functionality we needed. We could enter and change the weight value in that box. We have the variable assigned to it so we can retrieve that weight value in our program. But our Button didn’t do anything. We need to enable our program to sense when the button is pressed. Luckily, this is a standard function in Android so we can take advantage of the fact that there is a method in place to sense screen touches. We need to implement this in our code.

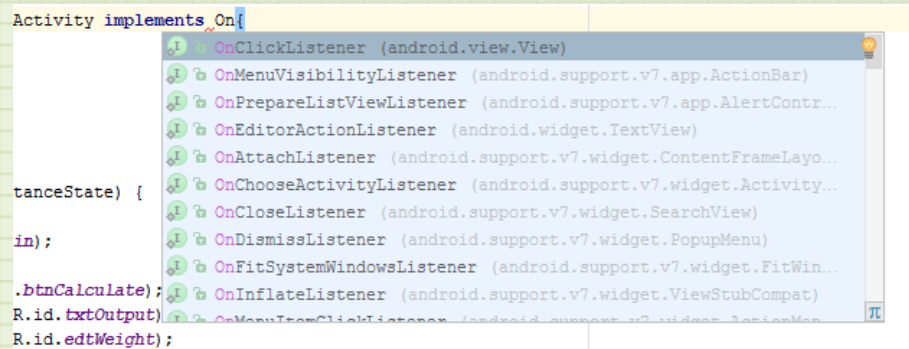
In your code, up where the MainActivity is defined, we are going to add an implementation to sense screen clicks. After the term “AppCompatActivity,” start typing the word “implements.”

A screenshot of a code editor showing the line `extends AppCompatActivity implements`. A blue suggestion box is open below the word `implements`, displaying the word `implements` as a suggestion. The text is in a monospaced font with syntax highlighting: `extends` is blue, `AppCompatActivity` is black, `implements` is pink, and the opening curly brace is blue.

Again, you should see that Android Studio is predicting what you want. When the little blue box shows up with the suggestion “implements,” click it or press Enter to accept it. The method that we want to implement is called **OnClickListener**. This is a piece of code that looks for touches on the screen and reports them to your program.

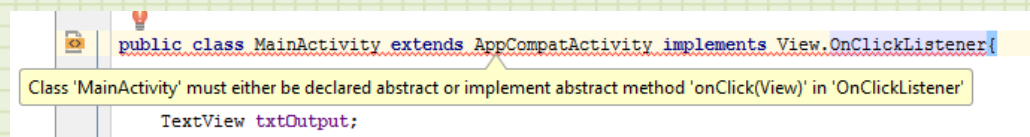
After the word “implements” start typing the term `OnClickListener`.

Again, you won’t get far before Android Studio suggests just what you are looking for. When you type enough to see `OnClickListener` show up in the suggestion box, click it or hit Enter to add it to the line of code.

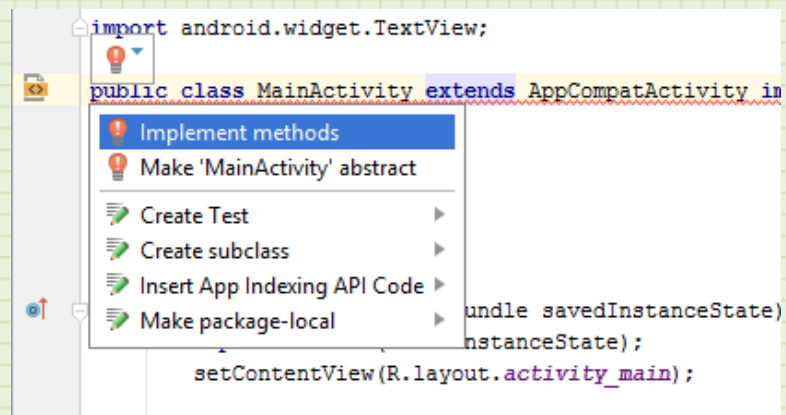


When you are done, you will see the method added to the line. But there is a problem.

There is a red underline that shows there is a problem on this newly created code. It is ok. We expect it. We just told the program to implement a method. That method needs to communicate screen touches to our program. But we haven't done anything to handle this communication. The red line is there to remind us that we need to implement this missing method. In fact, Android Studio will tell us this exact thing. If you click on the line, you should see something similar to the figure below.

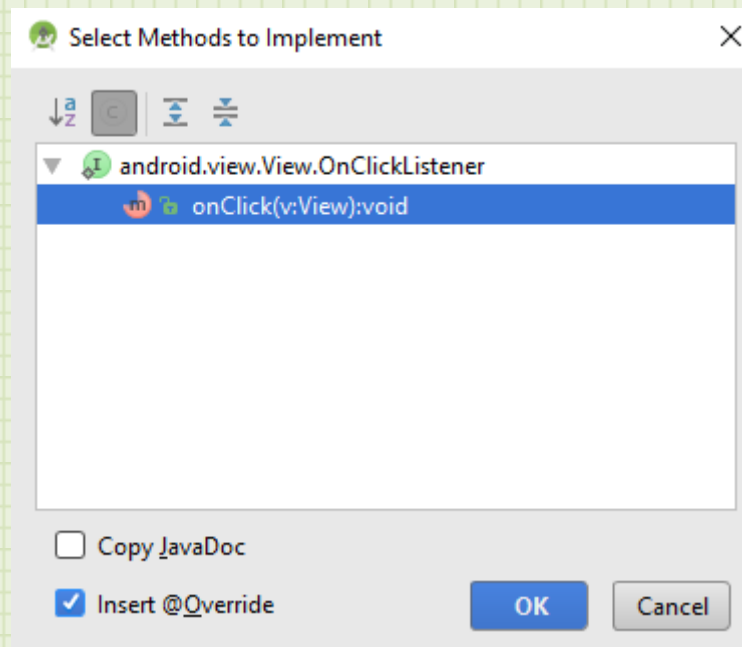


What this is telling us is that we just added an implementation (OnClickListener) but we didn't add a required method (onClick(View)). If you look over to the left on the line, there is a small drop down box. If you expand it, it will present you with some options to correct the problem.





Click on the option “Implement methods” to add the required method to your program. When the dialog pops up with the list of methods, select the `onClick` method (it is the only method there) and click OK.



Again, Android Studio to the rescue. Suddenly, the error is clear and a small code segment has been added to the bottom of your `MainActivity` file.

```
@Override  
public void onClick(View v) {  
    |  
}  

```

This is the code that runs when the screen is touched. The information the screen sends to your program is what “View” was actually clicked. A View is any widget or layout that can be interacted with on your screen. You may have noticed that a text box is actually called a `TextView`. The text box is actually a View that can be clicked or dragged and dropped or some other operation. The `onClick` method will allow us to determine which object was clicked so that we can do different things for different objects.

Now, we need to tell our program that we want the OnClickListener to listen for the Button to be clicked. We do this by registering our Button with the OnClickListener.

```
txtOutput = (TextView) findViewById(R.id.txtOutput);
edtWeight = (EditText) findViewById(R.id.edtWeight);

btnCalc.set
```

setOnClickListener (OnClickListener l)	void
setText (CharSequence text)	void
setAccessibilityDelegate (AccessibilityDelegate delegate)	void
setAccessibilityLiveRegion (int mode)	void
setAccessibilityTraversalAfter (int afterId)	void
setAccessibilityTraversalBefore (int beforeId)	void
setActivated (boolean activated)	void
setAllCaps (boolean allCaps)	void

Back up in your code where you defined the widget variables, add the code:

**btnCalc.setOnClickListener(this);**

Again, Android Studio will anticipate what you want and you will eventually see the suggestion to complete this line.

This line of code tells the OnClickListener to “listen” for our Button to be pressed and then run the code in onClick when it is pressed.

So, we currently have an app that has a Button that can be clicked and report it’s clicking to our program. We have an EditText to get the weight information from the user. And we have a TextView to display the results. Let’s write the code to calculate the weight on Mercury and display it to the user.

Enter the following code in the onClick method at the bottom of the file.

```
@Override
public void onClick(View v) {

    double dblWeight;
    dblWeight = Double.valueOf(edtWeight.getText().toString());

}
```

The first line creates a variable called **dblWeight** of type **double**. The keyword double represents a number that can be very large and have a decimal. It may be overkill for our use, but we are not concerned with conserving memory.

**Double** actually means double precision and takes twice as much memory to store as a **float** variable. Below are some of the number variable types in Java.

Variable Type	Number of bytes of Memory	Number Range
byte	1	-128 to 127
short	2	-32,768 to 32,767
int	4	-2,147,483,648 to 2,147,483,647
long	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,808
float	4	7 digit decimal number
double	8	16 digit decimal number

The first 4 variable types hold a signed integer number. The last two are for decimal numbers.

The next line of code takes the text that is in our EditText (edtWeight) and places it in this long variable we created (dblWeight). We are not allowed to assign different type values to variable so we have to convert our text to a double. When we use the function **Double.valueOf(something)**, we convert the something in parenthesis into a Double. This then allows us to assign it to our variable.

Remember that we set up the EditText variable so we could access its properties. When we use **edtWeight.getText()** we are running a function (getText) that is already built into our widget that returns the text in the EditText text field.

Now that we have retrieved the text, converted it to a double number, and assigned it to our double variable, we can use this to calculate the weight on Mercury.

To get the weight on Mercury, we just need to multiply the weight stored in dblWeight by the number 0.3772. That is pretty straight forward. What is not so straightforward is how to present this information to the user. We want it

to come with a label to identify it since we will eventually be displaying all of the planet weights. Enter the following line of code on the next line.

```
String outPut = String.format("Mercury: %.1f lbs", dblWeight * 0.3772);
```

The first word on the line is String. It is another variable type, but instead of a number, a String holds information that can contain numbers, letter, and symbols. In other words, it holds text. Strings have a lot of built in functionality so they are often good to use to get data ready to display. The next word (output) is our variable of type String that we will use to get our output ready. You may wonder why we can't just assign our double number directly to a String. We discussed this already. You cannot assign different type to each other. You must use some sort of conversion. To convert the String to the double above, we used the Double.valueOf() function. Here we are going to use another useful String function. The **format** function is a very versatile function that lets us combine words and numbers into a final output suitable for our application. It will also allow us to format the number the way we want. We can specify the number of decimal places to display numbers and whether to have leading or trailing zeros, and many other options. The format we ask this function to place our data in is held in the first part of the function: We want to format the output so it will have the name of the planet followed by the weight expressed to 1 decimal point. The following format string will do that for us.

```
"Mercury: %.1f lbs"
```

When the string contains text, it will just be copied to the output as is. When the format encounters a %, it knows that it will be inserting a number at that point. When we want to insert things in our output, we can use the following:

%s – add a string

%d – add an integer

%f – add a real (floating point) number

So, our line includes the term %.1f which adds a floating point number to the output. The .1 in between the % and the f tell it to add the number to 1 decimal point. The last part of the format string just adds the text lbs.

The format string is asking for a floating number so the second part of the function is a list of the data the format requires. We just formatted one floating point number so we just need to include one in the format function. It just so happens that we can do the multiplication we need in that variable list itself. The number we want to display is the weight (dblWeight) times 0.3772. So we just place that in the format list. The function will do that calculation before it applies the number to the format and sends it to the output. Again, the whole format line is:

```
String outPut = String.format("Mercury: %.1f lbs", dblWeight * 0.3772);
```

Now that we have a String (ouPut) that holds our desired output text, let's apply it to our output widget.

```
txtOutput.setText(outPut);
```

Here again, we are using our variable (txtOutput) that we assigned to our TextView. We are accessing the setText() function by using a period. That period allows us to access variables and functions associated with the widget type. The setText function expects a String to display. We send our String variable outPut.

```
@Override
public void onClick(View v) {

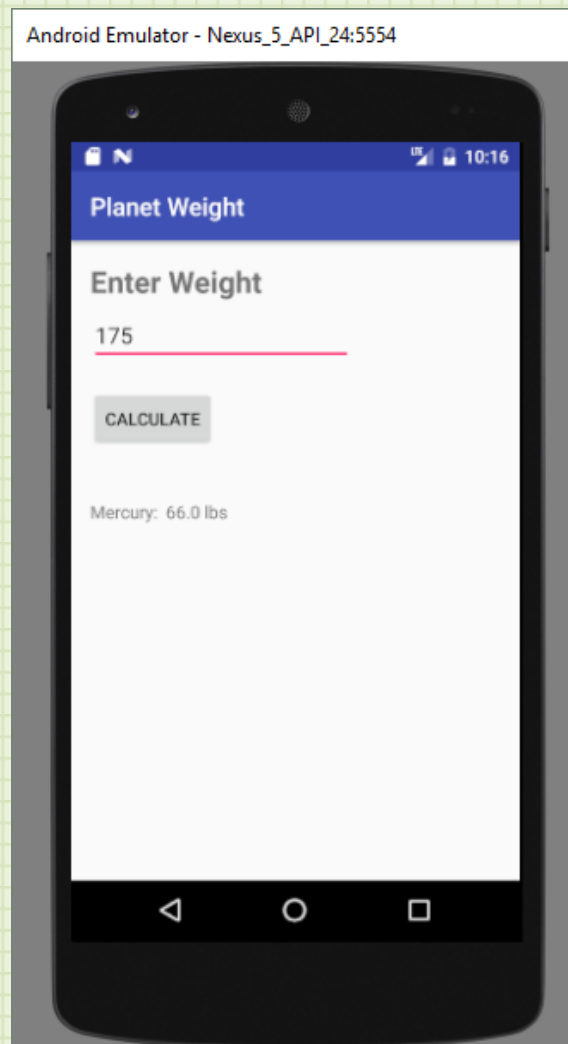
    double dblWeight;
    dblWeight = Double.valueOf(edtWeight.getText().toString());

    String outPut = String.format("Mercury: %.1f lbs", dblWeight * 0.3772);

    txtOutput.setText(outPut);
}
```

That is all of the code needed in onClick() to make a functional program.

Save your work and run this app as before. Now when you click on the Button, you will calculate the weight on Mercury and display it on the output TextView. Change the weight in the EditText and click the button again. This is pretty nice, but you can probably see some things you want to change before you offer this on the app store.



In the next section, we will look at completing this app to display the other planets, change the appearance to look more professional, and create an icon for the app.

Well done making it to this point. There is a lot of information here. Don't worry, this is an introduction. If you see that an app is made up of a graphical layout and some Java code, you already have a good feel for Android programming.

Let's look at some modification to make this a great app.

## Add the other planets

Let's include the rest of the planets in our app. Go back to the MainActivity.java file and go down to the onClick function where we formatted the text for the planet Mercury. Add the other planets in.

```
@Override
public void onClick(View v) {

    double dblWeight;
    dblWeight = Double.valueOf(edtWeight.getText().toString());

    String outPut = String.format("Mercury: %.1f lbs", dblWeight * 0.3772);
    outPut += String.format("\nVenus: %.1f lbs", dblWeight * 0.904);
    outPut += String.format("\nMars: %.1f lbs", dblWeight * 0.3783);
    outPut += String.format("\nJupiter: %.1f lbs", dblWeight * 2.527);
    outPut += String.format("\nSaturn: %.1f lbs", dblWeight * 1.064);
    outPut += String.format("\nUranus: %.1f lbs", dblWeight * 0.8858);
    outPut += String.format("\nNeptune: %.1f lbs", dblWeight * 1.137);

    txtOutput.setText(outPut);
}
```

You can see what we did. We assigned the original text to the variable outPut. The next line then adds the text for the next planet to the output variable. You can see in the format string that we add a “\n” before each planet’s name. This is the code for a line feed. This moves the printing of text to the next line. It is an ASCII code that is used by the computer to format the printing. You will see that it itself does not get printed. It is a simple way to add a line feed to a group of output text.

The conversion factors for the other planets are:

**Venus – 0.904**

**Mars – 0.3783**

**Jupiter – 2.527**

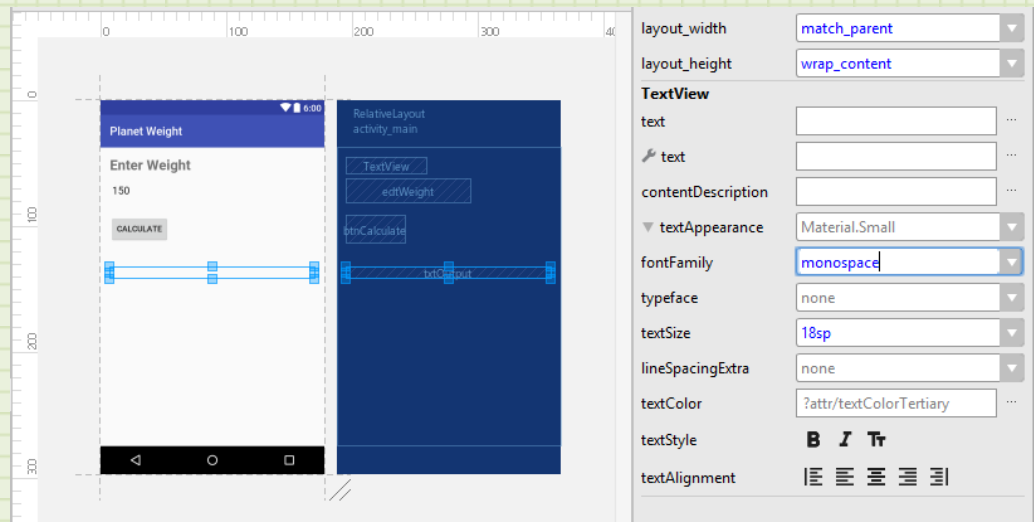
**Saturn – 1.064**

**Uranus – 0.8858**

**Neptune – 1.137**

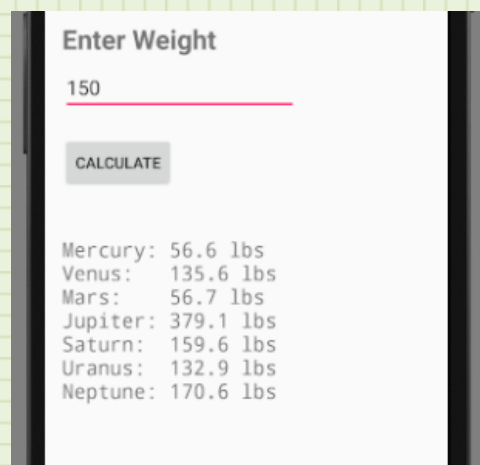
Notice also that we lined up the text in the format string to give us a neater look when they print. That sounds good, but the font we have is not a fixed size font so if we don't change that, the output will never be lined up right. Go to the activity\_main.xml file on the navigation tab to open the UI editor.

Click on the txtOutput widget on our layout. This will bring up the set of properties for us to edit.



Under the property `fontFamily`, choose the font “monospace” as the selection and change the `textSize` to 18sp. This will give us a given space between the planet name and weight and also display it a little bigger.

Save and run your app.



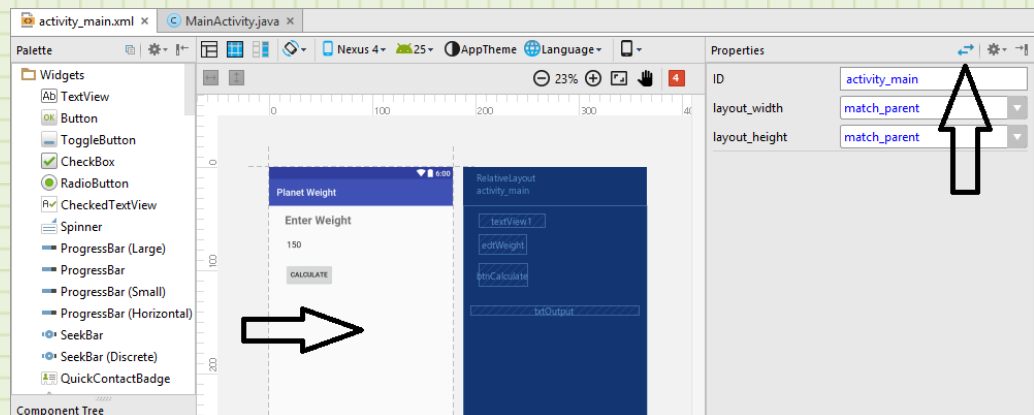
This looks pretty good. But let's customize it a little more. Let's change the background color. And let's move the output display a little to the right. And let's add a custom icon.

We'll take these changes one at a time so you can see how to do each one.

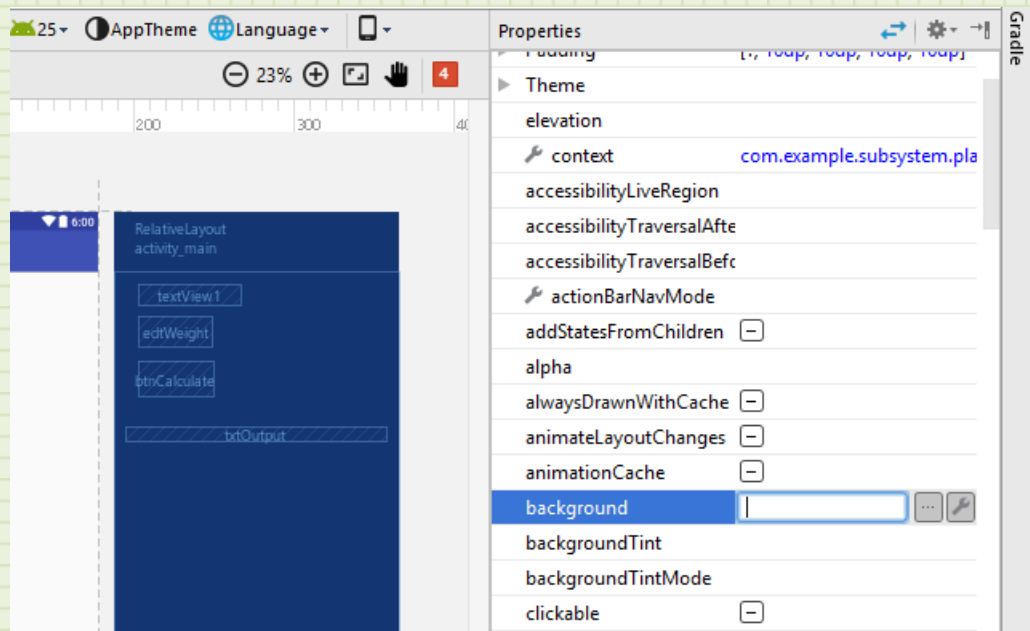


## Create a Custom Color

To change the background color, we need to change the color property of our background. Open the file **activity\_main.xml** to access the app layout.

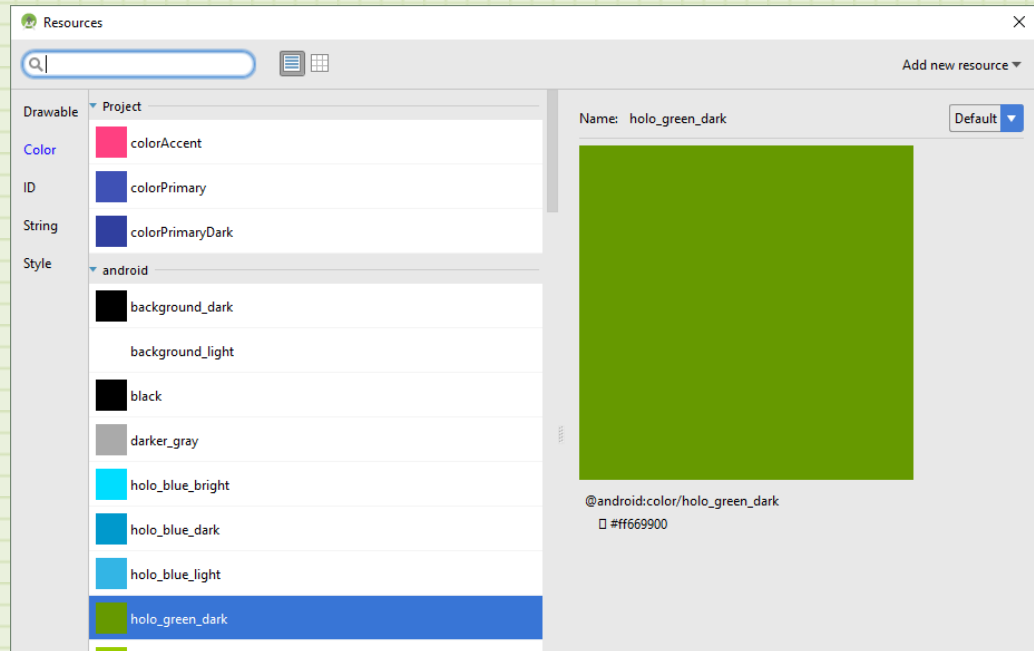


Click on the white space on the left layout picture (left black arrow). This will select the background widget (which is our Relative Layout). This will call up the properties for that widget in the right panel. By default, this is only a short list of the most common properties that you may want to change. The icon in the top right (pointed to by the right black arrow) opens up the detailed list of properties. Click the icon to shift to the detailed list.



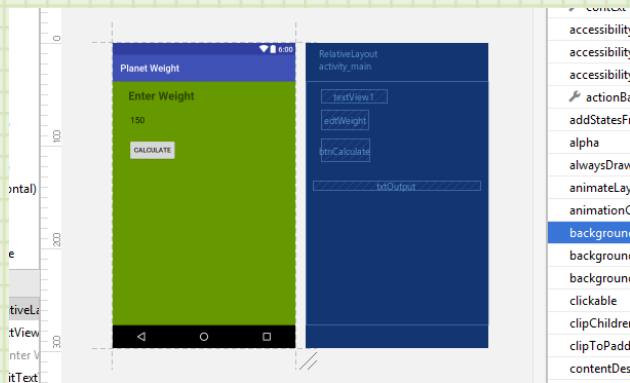
Go down the list until you come to the **background** property. Click on it to bring up the selection box. The first button to the right allows you to choose the color from all the colors available to your program. Click on it (it is the button with ... on it).

This brings up the color dialog. The far left column shows the different places that color information is contained in the project. Click on the item “Color.”



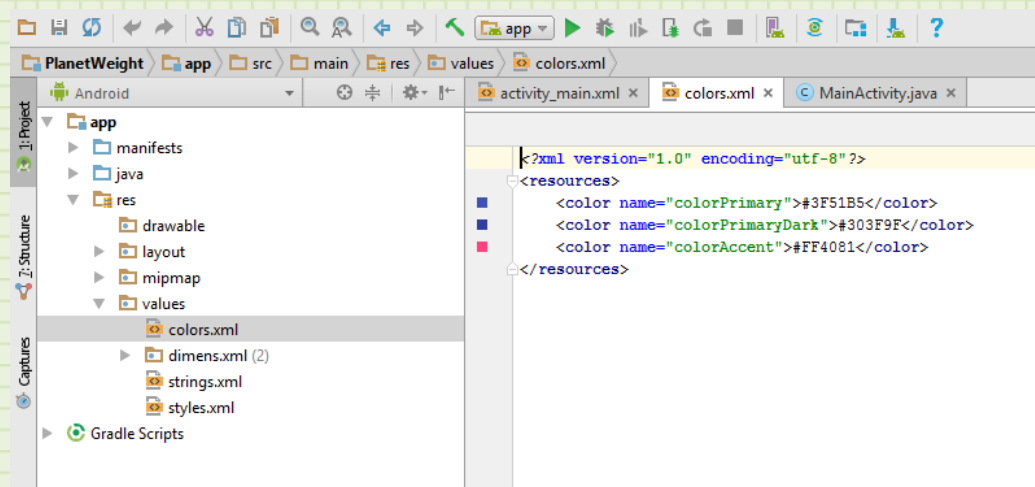
This will bring up choices from various places in the project. The first section has the colors you have defined in your app. The three listed were placed there by the wizard when we started out. The next section shows colors available that are standard to the Android system. Scroll down and select the **holo\_green\_dark** color and click OK.

The background of your app now has a pretty green color.

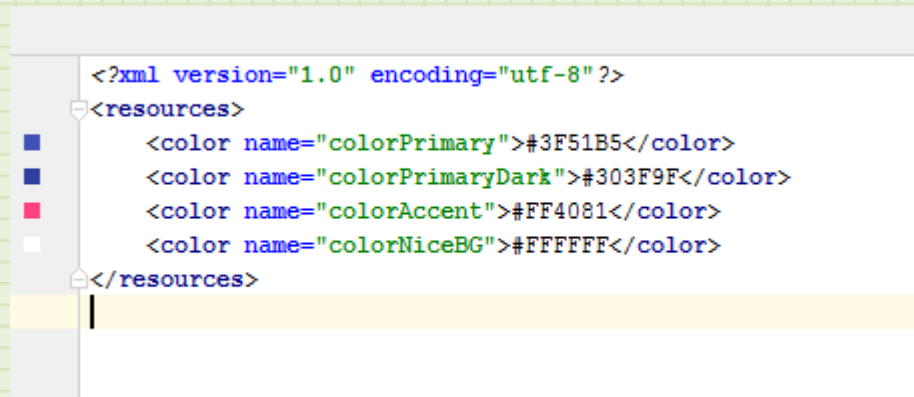


Let's say you are not a fan of green. Here is how to add a custom color to your project.

Back on your project page, in the left project navigation window, expand the **app** folder, then the **res** (resources) folder, then the **values** folder. You will see a couple of files there. Double-click the **colors.xml** file. This will bring up the custom project colors.

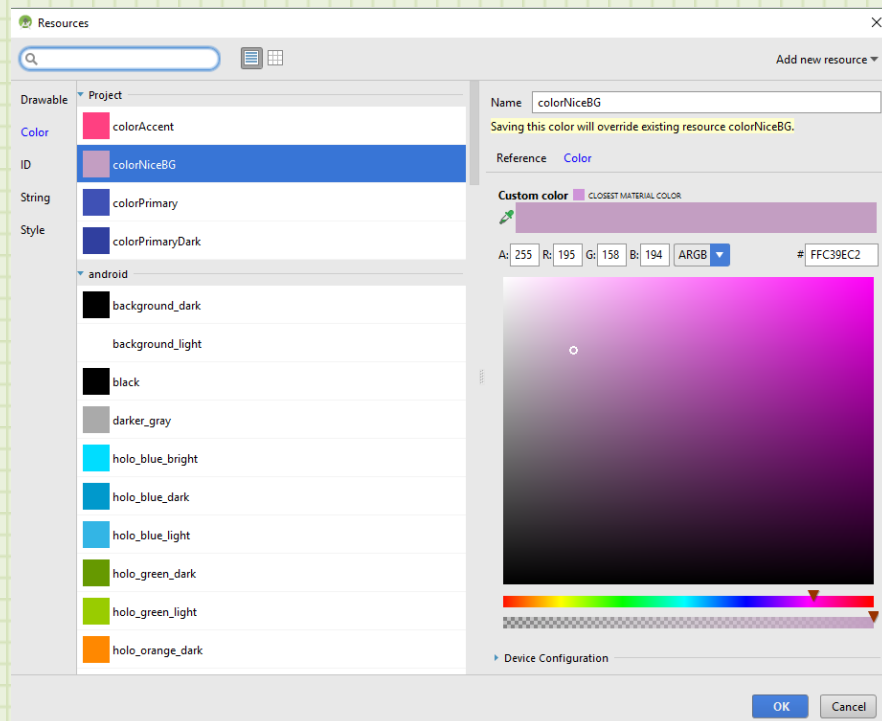


You can see the current entries. These were the colors available in the color dialog when we changed the color of the background. If you look at how the color is defined, it is given a name and then a hexadecimal code that represents the amount of primary colors that are used to create it. There are many online resources that will generate the desired color. We are going to take advantage of Android Studio's color picker. After the last color entry, add a new custom color. Call it **colorNiceBG** and assign it a value of **#FFFFFF**.



You will see that a small box of the color shows up in the column to the left. The color you entered was white.

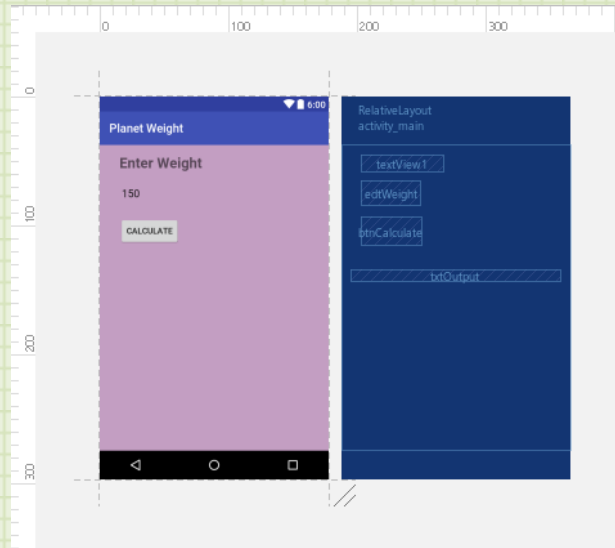
Now go back to the layout, click on the background, and find the **background** property in the list as you did before. Click the button with three dots to open the color dialog. Now you see our custom color listed in the project colors. There is also a color picking tool on the right.



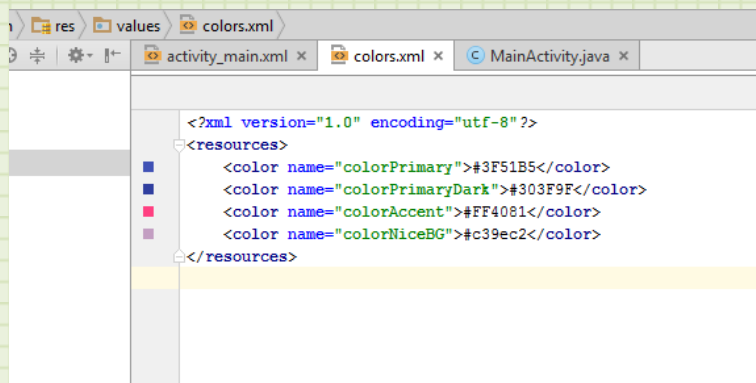
Here, you can create any color you want and it will update our custom color with the new value. Move the slider along the rainbow strip at the bottom to find a base color you like. Then grab the small white dot in the upper left

corner and drag it to the exact shading you want. In the above image you can see we chose a purple base color and then lightened it so the black text in our app shows up well. The color you are currently selecting is shown in the box under the words **Custom color**. Find a color you like and select OK at the bottom of the dialog.

After you do, you will see that your background now has a truly custom color.



If you look back at the colors.xml file, you will see that Android Studio has updated our color with the new value. You can see the sample of it to the left of the line of code that defines it.

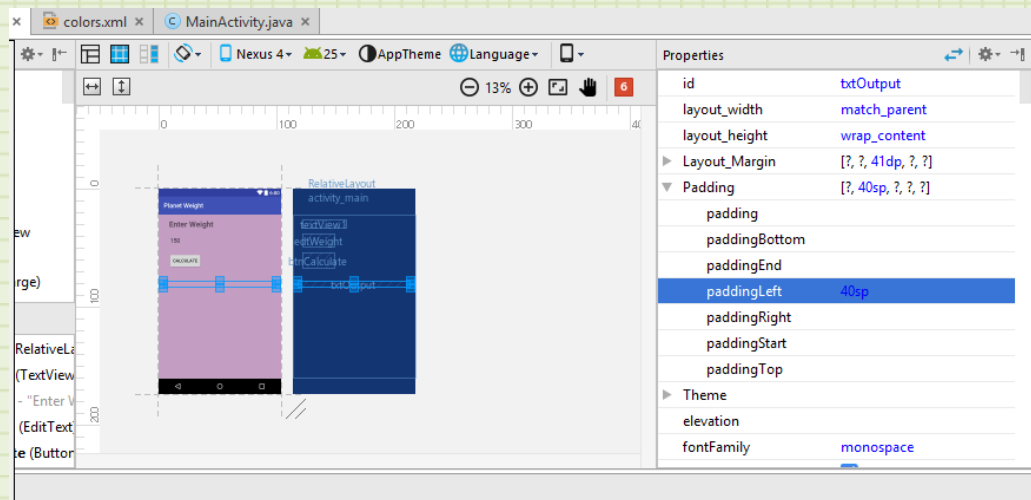


With this method, you can create custom colors for all of the components in your app.

## Add Padding to an Object

The second thing we want to do to continue to fine tune our app is to add some padding to the output display box to give it a feel of being more centered on the display. There are a couple of ways to do this. But we will add some padding to the widget. Padding is space that you can add to the inside borders of a widget.

In the layout file, click on the **outPut** TextView widget. That will bring up the properties.



Scroll down the list of properties until you get to the Padding property. Click the triangle to expand it. You will see all of the different places you can add a little padding. In the **paddingLeft** box, type in **40sp**. This represents 40 pixels and the “sp” is a code that uses the text scale for the size of the pixels.



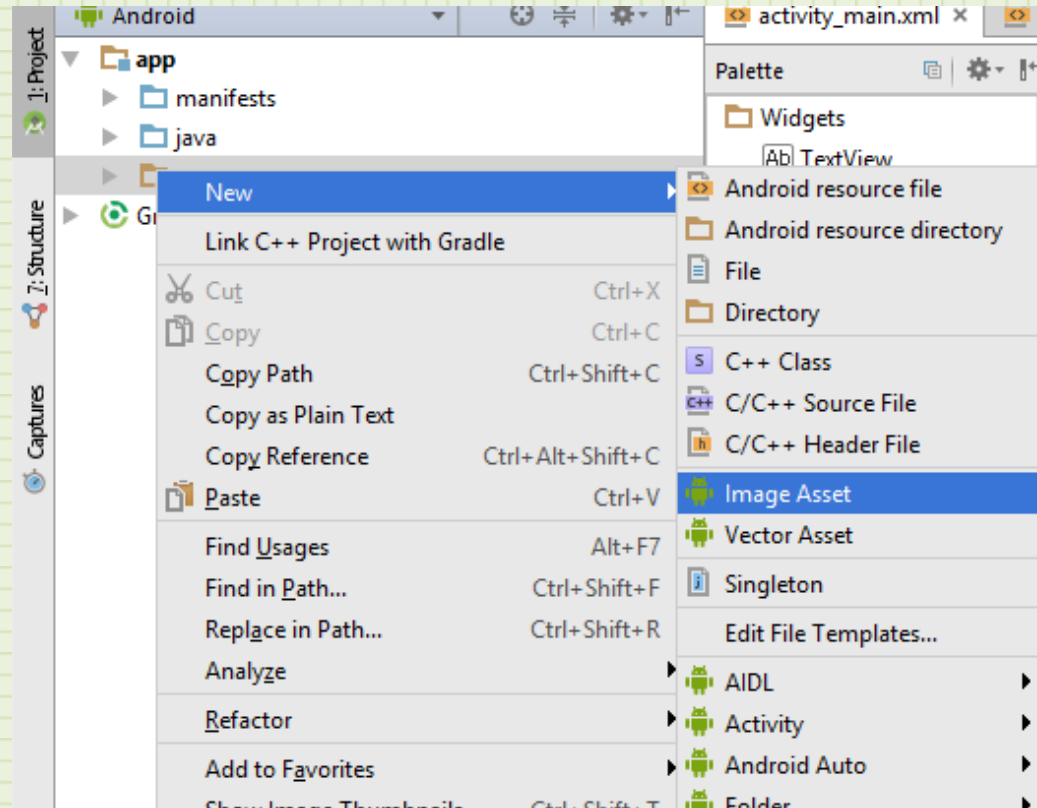
When you run the app now, you will see that the output text looks a little more centered on the display.

Padding and Margins are ways to help format the look of you UI.

Now, let's add an icon.

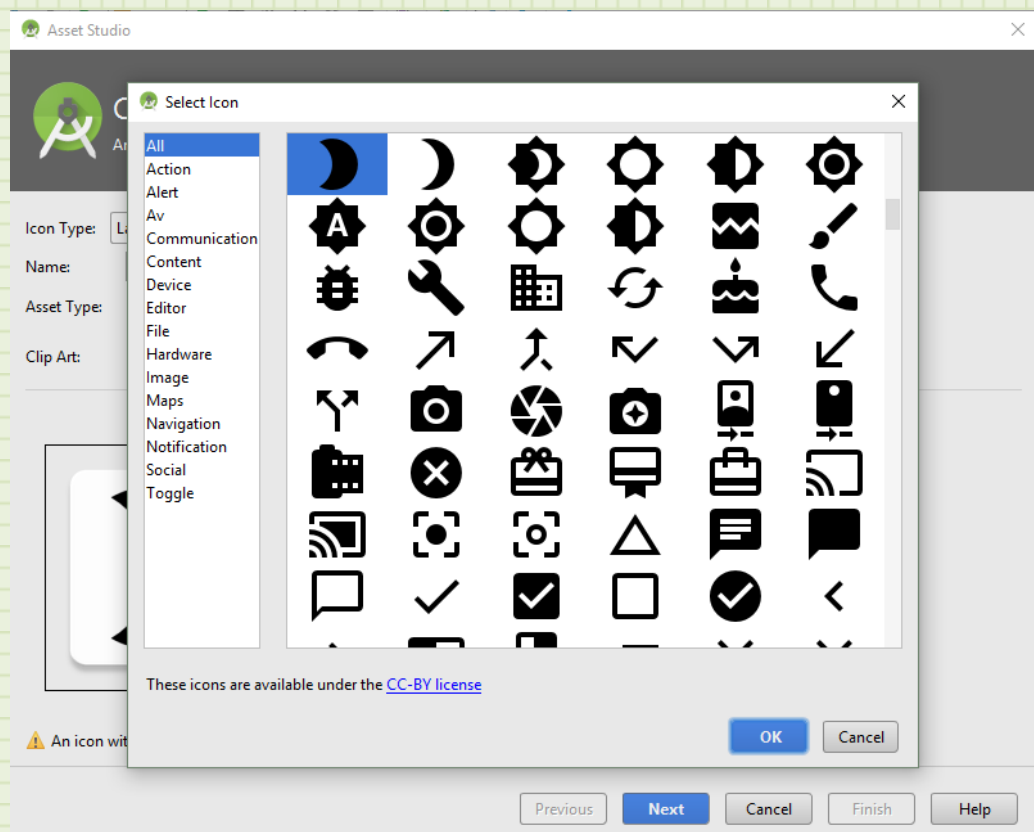
## Adding a Custom Icon

When we initially created the app, the wizard assigned it a standard Android icon. But we want our app to have a custom icon. With Android Studio, it is easy to do this.



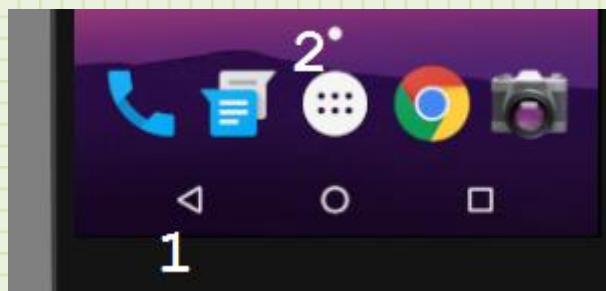
At the main screen, on the left in the project navigation pane, expand the **app** folder to reveal the subfolders. Right click on the **res** folder to bring up a list of options. Select **New** and then select **Image Asset** from the menu.

This will open the Asset Studio. Here, you will be able to create an icon for your app. The program will let you choose an image, clip art, or text for your icon. Asset Studio comes with a selection of icons so let's use one of these. Click on the box next to the words Clip Art. This opens a clip art selection dialog. On the left is a list of categories to limit your search, but for now, scroll around and look at the different clip art choices. Find one that you want to represent your app.



Click on your selection and then click the **Next** button on the bottom of the dialog. The next dialog reminds you that you are overwriting the current icon. Click Finish at the bottom of the screen. You now have a custom icon for your app. Let's take a look at it in action.

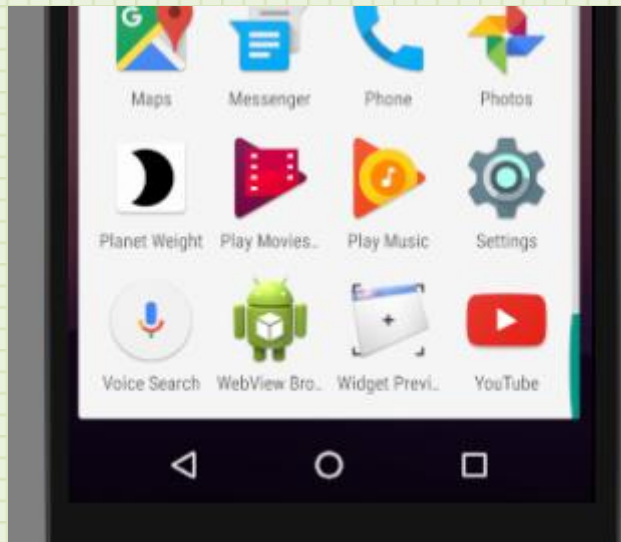
Run your app on the emulator as before. After the app starts, click the small triangle at the bottom of the emulator screen (labeled 1 in the picture).



Then, click on the white circle with the dots in it (labeled 2 in the picture). This will bring up a listing of all the apps on the emulator.



Scroll down until you see your app. Notice that the icon is the new one you have created.



You now have a functional, customized Android app. From here, you could continue to modify and add functionality to it or could even start a brand new project. The appendix has some great resources to continue learning about Android, Java, and programming concepts. Google also has a lot of information to help you take any app you create and load it up to the Google Play Store. See the Appendix for these sources and more.

We hope this lesson has shown you that the world of app writing is not the domain of the professional computer programmer alone. With some time and practice, you too can generate useful apps and post them to the app store and even generate revenue from selling apps or servicing ads. The intention of this lesson was to load up Android Studio, get familiar with the interface, UI basics, and some simple Java programming. Hopefully, you feel confident to go online and find some more tutorials and lessons to allow you to build on what you have learned.

## Appendix A: Additional Resources

The best resource for Android Programming is Google's own site:

**[developer.android.com](https://developer.android.com)**

For more introduction information:

**[developer.android.com/training/](https://developer.android.com/training/)**

San Jose State University has a free course on Java programming:

**[www.udacity.com/course/intro-to-java-programming--cs046](https://www.udacity.com/course/intro-to-java-programming--cs046)**

Google gives extensive information on how to become a registered developer and post apps on Google Play.

**[developer.android.com/distribute/googleplay/](https://developer.android.com/distribute/googleplay/)**

Tutorials Point is a good site for a lot of Android programming examples:

**<https://www.tutorialspoint.com/android/>**

## Appendix B: Loading an App on your Android Device

This section is provided to help you load your new app on an actual Android device. Since this is more complicated due to the variations of Android devices and the level at which your carrier may lock your phone, it is not guaranteed that you will be successful. That said, a lot of phones work with little effort.

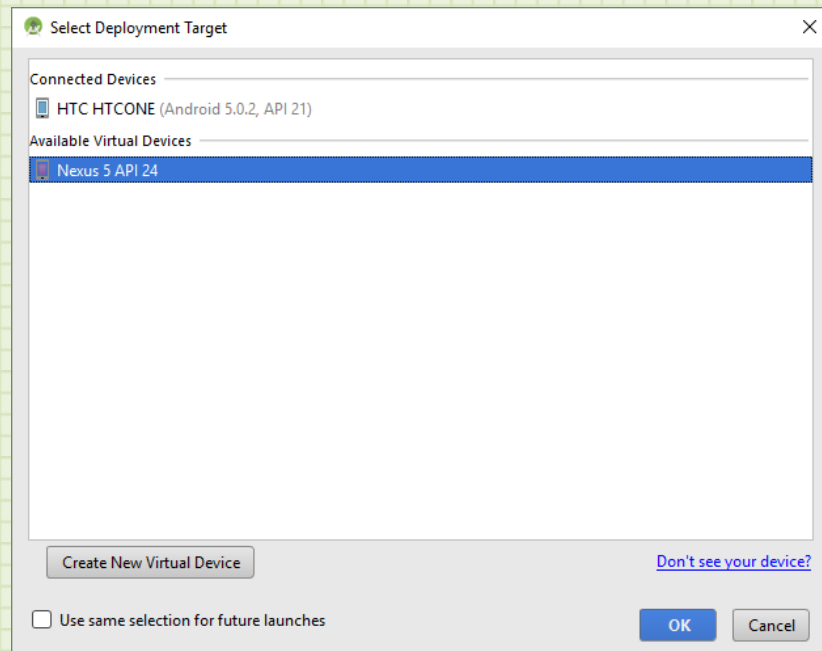
The goal is to have Android Studio see your actual phone as a legitimate place to load your app on to. To do this, the phone must be recognized by your computer and must be able to enter developer mode.

First, you have to enable USB debugging. **Select Setting->Developer Options.**

Note: If you don't see this option (it is hidden on Android 4.2 and later), go to **Settings->About Phone** and find where it lists the **Build number**. Tap this area at least 7 times. When you return to the Settings menu, Developer Options will now appear.

Make sure you have enabled **USB Debugging** in the Developer Settings. Plug your phone into your computer. If your computer has a compatible driver, your phone will be recognized after the driver is loaded. If your phone displays messages, click the options to allow access.

If all goes well, go to Android Studio and click to run your app. The normal Emulator window will pop up to ask you what you want to run the app on. Your phone should appear on the top under Connected Devices. If it does, just select it. Android Studio will load the app on the phone.



If you have any issues doing this, you may need to load drivers or change a configuration. See the following site:

**[developer.android.com/studio/run/device.html](http://developer.android.com/studio/run/device.html)**

for more complete information on troubleshooting problems.

# Appendix C: Code Listings

Here is a listing of the code you should have in the Java and layout files:

MainActivity.java

```
package com.example.subsystem.planetweight;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity implements View.OnClickListener {

    Button btnCalc;
    TextView txtOutput;
    EditText edtWeight;

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

        btnCalc = (Button) findViewById(R.id.btnCalculate);
        txtOutput = (TextView) findViewById(R.id.txtOutput);
        edtWeight = (EditText) findViewById(R.id.edtWeight);

        btnCalc.setOnClickListener(this);

    }

    @Override
    public void onClick(View v) {

        double dblWeight;
        dblWeight = Double.valueOf(edtWeight.getText().toString());

        String outPut = String.format("Mercury: %.1f lbs", dblWeight * 0.3772);
        outPut += String.format("\nVenus:   %.1f lbs", dblWeight * 0.904);
        outPut += String.format("\nMars:    %.1f lbs", dblWeight * 0.3783);
        outPut += String.format("\nJupiter: %.1f lbs", dblWeight * 2.527);
        outPut += String.format("\nSaturn:  %.1f lbs", dblWeight * 1.064);
        outPut += String.format("\nUranus:  %.1f lbs", dblWeight * 0.8858);
        outPut += String.format("\nNeptune: %.1f lbs", dblWeight * 1.137);

        txtOutput.setText(outPut);

    }
}
```

## activity\_main.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/activity_main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="com.example.subsystem.planetweight.MainActivity"
    android:background="@color/colorNiceBG">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_below="@+id/btnCalculate"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true"
        android:layout_marginTop="41dp"
        android:id="@+id/txtOutput"
        android:textSize="18sp"
        android:fontFamily="monospace"
        android:paddingLeft="40sp" />

    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Enter Weight"
        android:textSize="24sp"
        android:textStyle="normal|bold"
        android:layout_alignParentTop="true"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true"
        android:layout_marginLeft="17dp"
        android:layout_marginStart="17dp" />

    <EditText
        android:layout_width="100sp"
        android:layout_height="wrap_content"
        android:inputType="numberDecimal"
        android:ems="10"
        android:id="@+id/edtWeight"
        android:text="150"
        android:layout_marginTop="16dp"
        android:layout_below="@+id/textView1"
        android:layout_alignLeft="@+id/textView1"
        android:layout_alignStart="@+id/textView1" />

    <Button
        android:text="Calculate"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="17dp"
        android:id="@+id/btnCalculate">
```

```
        android:layout_below="@+id/edtWeight"
        android:layout_alignLeft="@+id/edtWeight"
        android:layout_alignStart="@+id/edtWeight" />
</RelativeLayout>
```