IN721 2019 Practical 1.1 – First App and using an Android Virtual Device

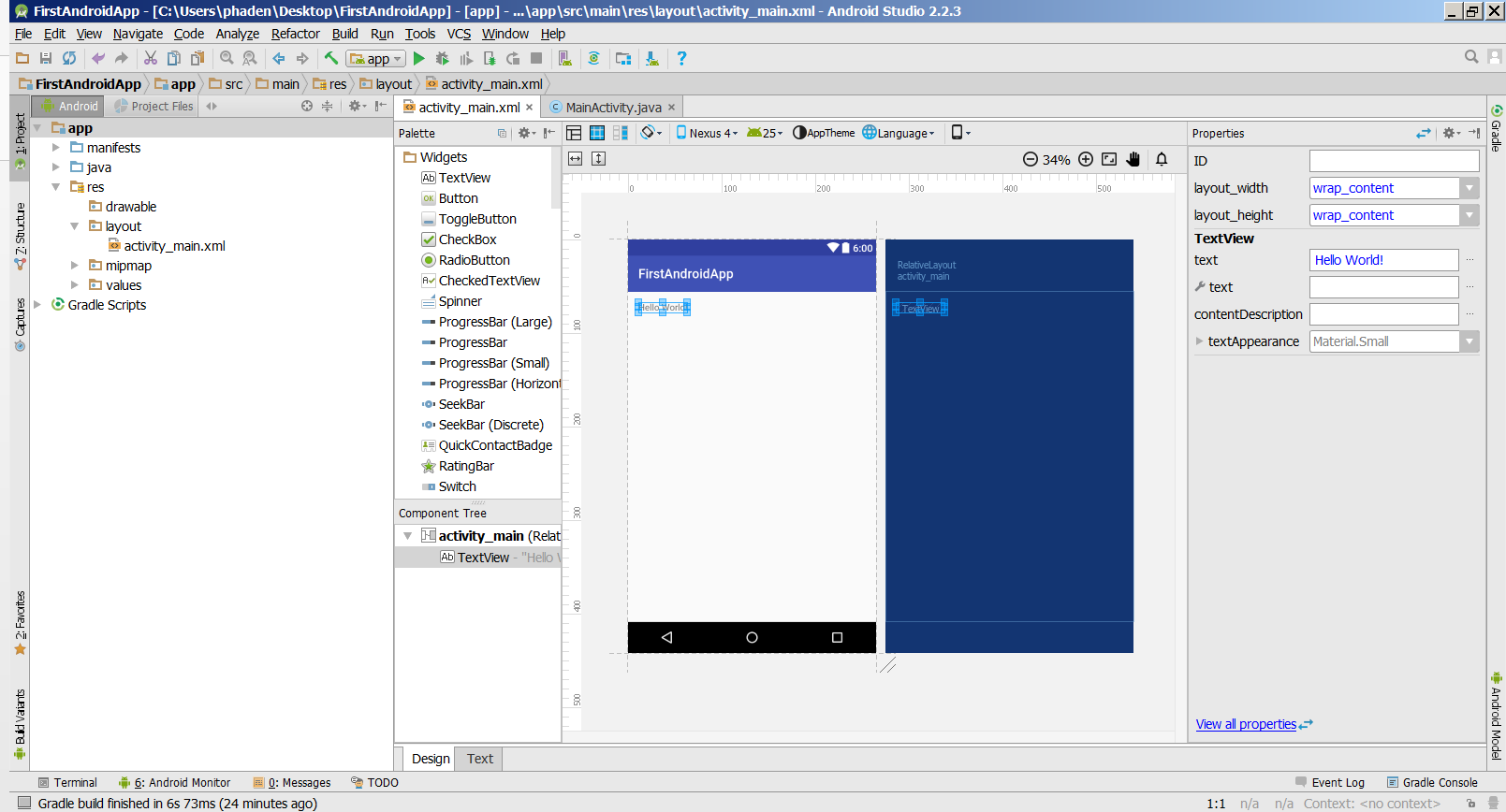
In this practical you will use Android Studio to build a basic Android application containing a single activity with two screen controls (called “widgets” in Android world). You will launch your app in the Android Virtual Device software emulator. The primary purpose of this exercise is to confirm that your machine installation and configuration are correct for Android development.

1. **Create your app**

Follow the process shown in the lecture PowerPoint to create an Android Studio Project with a single empty activity. Please be careful to observe all requirements for naming conventions, target APIs, etc.

1. **The Layout**
   1. Design view

On most machines, your created application will open with Android Studio in Design (WISYWIG) view, something like this:



If this is not what you see (or at any time you need to navigate to this view), double-click on /res/layout/*activityName*.xml in the Project Explorer pane, and click on the Design tab at the bottom of the editing pane (circled in the image above).

The Design view shows approximately what your app will look line when running on a generic Android phone. On this phone, the tool bar (with the mock power meter and time display) and the app's header bar (showing the app name) are part of the default *app theme*. These screen elements are not directly programmable, but the entire app theme can be changed from within Android Studio (left as an exercise).

The text control containing "Hello World!" is provided on creation of your empty Activity, as a simple example of the Android screen widgets.

Look at the Component Tree pane (usually located below the Widget Palette). The Component Tree shows the elements contained in the Layout. The Layout is described in XML (we will look at the XML source file shortly), and is therefore hierarchical. That is, it is an inverted tree-like arrangement of elements contained in other elements. This is similar to the structure of an HTML page, where you might have, for example, a list contained in a paragraph contained in a div.

In the Component Tree you can see that you have two screen elements: a RelativeLayout and a TextView. Each of these is an instance of a class defined in the Android libraries. The two elements can be seen in the XML source file, and can be accessed as objects in the code behind. We will learn both of these techniques this week.

RelativeLayout is a ***non-visual*** screen control. It is one of a class family of such controls, which also includes LinearLayout, FrameLayout and GridLayout. The Layout class objects do not appear on the screen at runtime. Rather they are containers that manage their contents in different ways (similar to a <div> in HTML). For example, LinearLayouts put their content in a single row or a single column; RelativeLayouts allow free placement; GridLayouts are a grid, etc. You can have multiple Layout class instances in a single screen, and Layouts can be freely nested.

The TextView is a simple text display element, equivalent to a .NET Label. The Component Tree uses indentation to show that the TextView object is contained within the RelativeLayout.

At the right-hand side of the screen is the Properties pane. (If you don’t see it, click on the “Hello World” TextView.) The default view of the Properties pane shows just the most common control properties. Click the “View all properties” link at the bottom of the pane, or the bidirectional arrows toggle, to display all of the available properties. We will see that when in Design view, we can modify many of the properties of our screen elements using the Properties pane, just as we do in Visual Studio. Changes we make in this way will be automatically reflected in the supporting XML source file.

* 1. Text view

The Design view is very useful for getting a sense of what your app will look like, and certain editing and design operations are easiest to do from Design view. However, it is, in actuality, simply a visual representation of an associated XML file that is created when you create the Activity. What really determines how the screen looks at runtime is this XML. To see the underlying XML, click on the Text tab at the bottom of the screen. This tab is directly to the right of the Design tab. For this example app, the XML is (your tools:context attribute will be different):



This is syntactically correct XML -- a self-closing <TextView> element, contained within a <RelativeLayout> element.

Note however, that, unlike many XML contexts, all of the information about these elements is expressed with attributes in the opening tag. The <RelativeLayout> for example, contains ten different attributes, all with lengthy value strings, before its closing right angle bracket.

This is the Android way. Do we like it? No. Is it ugly and hard to read? Yes. Will we learn to do it anyway? Yes.

We will look in detail at all these various tags, attributes and values as we go. For now, just note that everything you set in the Properties window in the Design view will appear in this XML file, usually as an attribute.

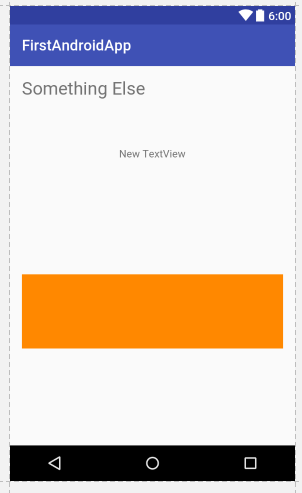
Note that the values of these attributes are all expressed as literal strings, but the interpretation of those strings differs for different attributes. In some cases, the strings are, in fact, the literal value of the attribute. For example, android:text="Hello World!" is the literal string contents of the TextView control. In other cases, the string is a variable value that is parsed by the system. For example, android:paddingLeft="16dp" causes the system to insert 16 density-independent pixels of padding on the left border of the TextView[[1]](#footnote-1). Finally, some attribute values are Android system constants. For example "match\_parent” is an Android enum value, which indicates that the control’s dimension should equal that of its containing (i.e. parent) control. Later we will see a fourth case, where an attribute value refers to a ***resource*** of the project, such as an image file or user-defined constant.

### TASK: Editing the Layout XML Source File from the Text and Design views

* Make sure you are in Text view.
* Change the text your app displays from “Hello World!” to something else, by manually modifying activity\_main.xml (or whatever your XML layout file was named during project creation). Use a literal string, as described above (for now, ignore any warning messages that appear). You should see the contents of the TextView change in the preview pane to the right of the edit pane (open this by clicking the Preview tab, if necessary).
* **Switch back to Design view**. Confirm that the change you made in the XML file is reflected in the Design view.
* While in Design view, use the Properties pane to increase the text size of the TextView to 24. You will need to 1) select the TextView control by clicking on it, and 2) find the textSize entry in its full list of properties.
* To change the value of the textSize property, click in the editable area beside the “textSize” item and type a value. (Alternatively, you can click on the ellipses to the right of the editable area to open the Resource Chooser. For this exercise, we will type directly into the editable area.)
* If you type "24" and hit return, the value will change to **24sp**. We saw earlier that we use **dp** (density-independent pixels) for screen widget dimensions like margins and padding to make our controls adjust to the screen density of the device on which the app runs. For font sizes, however, we need to use a different Andoid unit -- **sp**, scale-independent pixels. Scale-independent pixel size depends not on the characteristics of the screen, but on the font-size preferences that have been set by the user. For example, visually impaired users may increase their system-wide font size settings, and your app must respond to this preference. Using sp for your controls insures this. It is recommended that you always use sp (scale-independent pixels) for font sizes and dp (density-independent pixels) for spatial properties. (More details available on android.developer.com.)
* Set your textSize to 24sp.
* **Return to Text view.** How has the change you made in the Properties window been reflected in the underlying XML source?

### TASK: Adding Screen Elements in Design View

* **Return to Design view.**
* While in Design view, you can add more controls to your screen layout by drag-and-drop from the Widget Palette to the screen ***or to the Component Tree***. It is sometimes physically easier to place controls using the Component Tree. As you become familiar with Android Studio, you will be able to decide which technique you prefer.
* Add another **TextView** to your layout by dragging from the Palette. Note the gridlines which help you centre and align.
* Add an **ImageView** (equivalent to a .NET PictureBox) below your new TextView. You will need to scroll down in the Widget Palette to find the ImageView.
* When you drop the ImageView on the screen, a Resources window will pop up, requiring you to select an image file and offering you a number of system defaults. (To display your own image file in an ImageView, it must first be imported as a project resource. We will see how to do this in a later practical.)
* For now, configure your ImageView to appear as a solid coloured rectangle as follows:
* In the Resources window, select Color (at the left of the window), and click on a colour from the list provided.
* Your coloured rectangle **will not** appear in Design view. This is because the width and height of ImageViews adjust, by default, to the size of their contents, and your ImageView is currently empty.
* Modify the layout:height and layout:width properties of your ImageView to make the layout look approximately like this:



* Frequently move back and forth between the Design view and the Text view to see how your layout is expressed in the underlying XML. Note especially how, when in a RelativeLayout, the position of a screen control is described ***relative to*** other controls in its layout.

1. **Running the app**

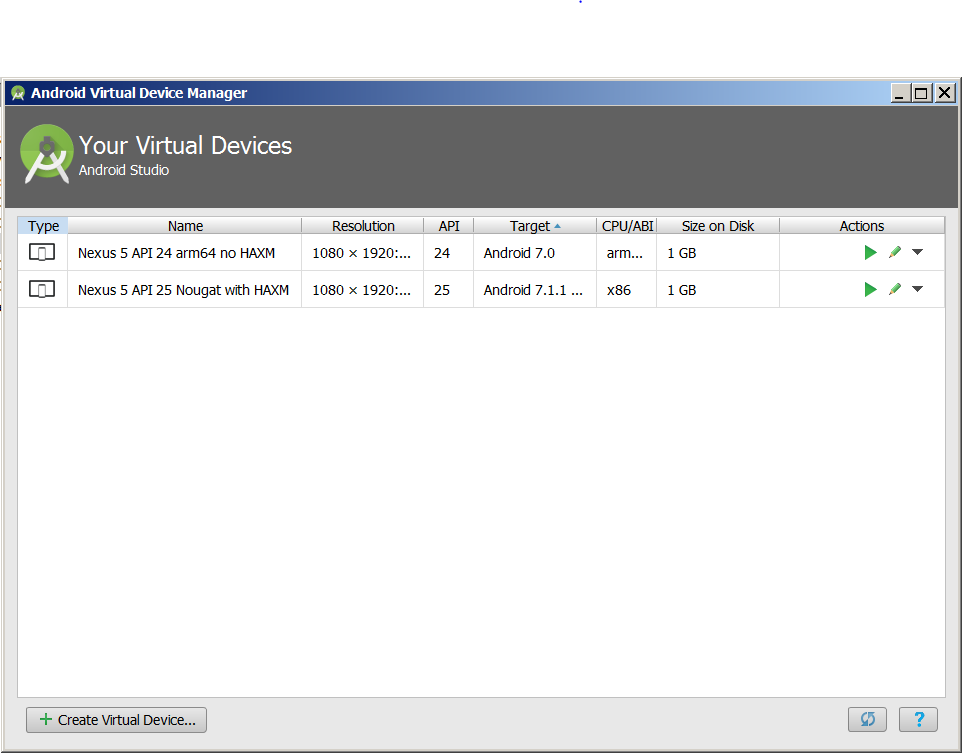
The empty Activity which Android Studio created for you has sufficient code behind to correctly launch and display on an Android device. In our next practical, we will start modifying that code; today, we will use this simple app to learn how to work with the Android Studio software device emulator[[2]](#footnote-2). Eventually, we want our apps to run on real phones and tablets, but during development, we can use the software emulator to test our code on the widest possible range of devices.

To run your app in the emulator, you first define an Android Virtual Device, specifying the hardware and software characteristics you wish to emulate. When you run (or debug) your project, you can select that device as the launch target. The images below show two different AVDs running on my machine. On the left is a 1080x1920 xxhdpi Nexus 5 running Marshmallow; on the right is a 480x800 hdpi Nexus 1 running Ice Cream Sandwich.

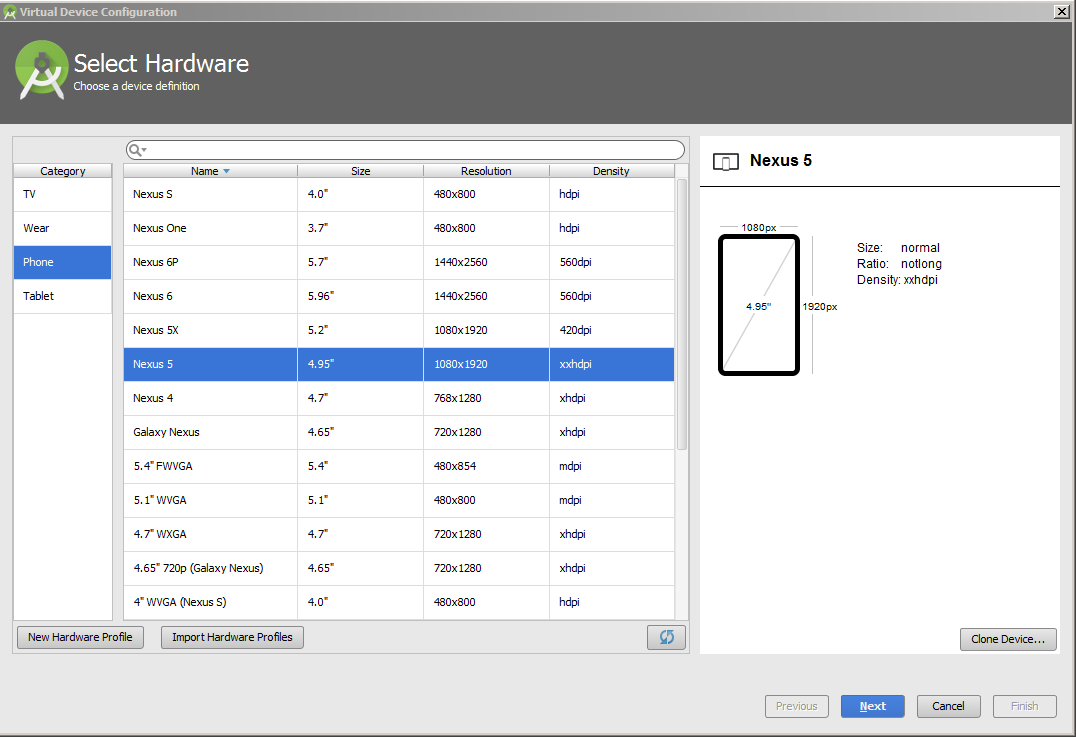


# TASK: Define an AVD

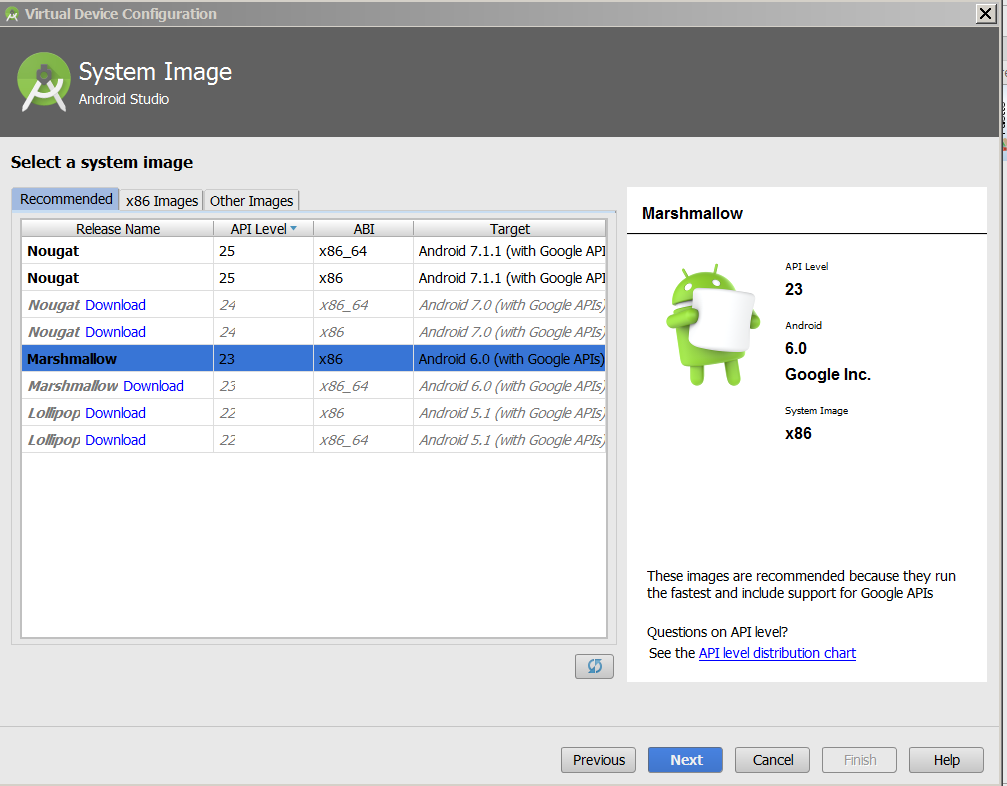
* To create an AVD, select Tools->Android->AVD Manager. The image below shows my AVD Manager, with some AVDs I have already created. A standard installation of Android Studio will have a single default AVD already created, but you can have as many different AVDs as you wish.



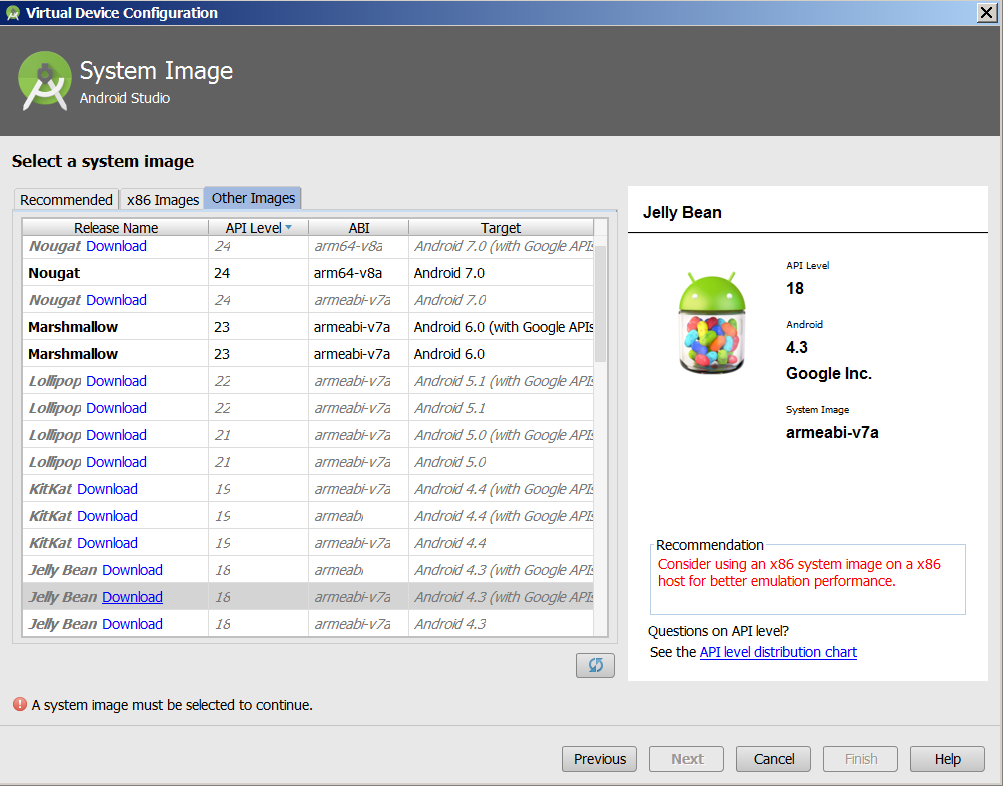
* Begin by clicking the Create Virtual Device button at the bottom left of the AVD Manager window. This will bring up the Select Hardware screen.



* The AVD Manager is quite complex and provides features for specifying exactly the hardware you wish to simulate. You may wish to explore this tool on your own. For now though, we will create an AVD using one of the pre-built configurations.
* Select Nexus 5 from the list of Device Definitions, then click Next. You will be presented with the System Image screen. The System Image defines the version of the Android operating system and of the API that the AVD will emulate. The figure below shows an example AS installation that has three different system images available. The other system images shown are suggestions, with associated download links:



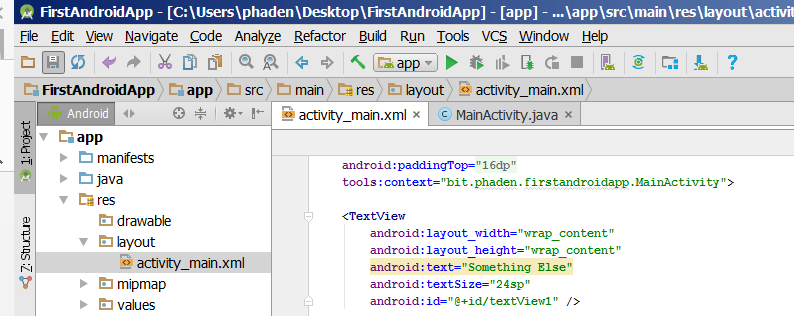
* Each System Image specifies a version of Android, an API level, an ABI (Application Binary Interface – the device CPU and machine instruction set) and the build target. The student machines will have a default set of images, but there are dozens of different combinations available. To see all the images that can be added to the running version of AS, click the various tabs at the top of the list. Click the Download link beside any image to install it.



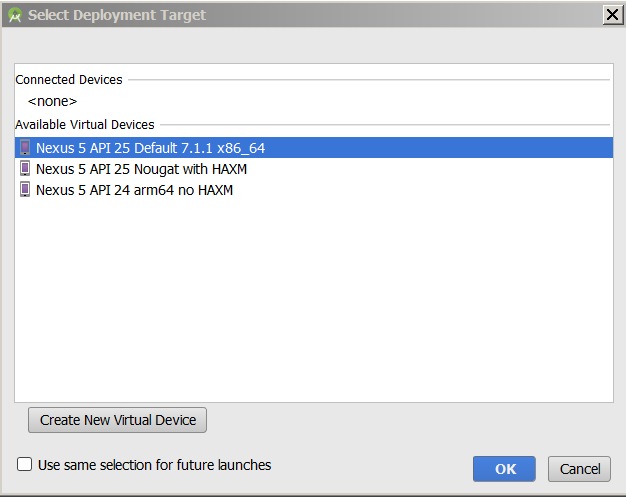
* For this practical, select any of the available System Images. You will be presented with a Verify Configuration screen. Change the AVD Name (so you can distinguish this new AVD from any existing one on your machine) and click Finish to proceed. You will be returned to the main AVD Manager window (the first figure in this task) and your new AVD should appear in the list.
* Each AVD entry has some Action buttons at the far right: launch, edit, and a drop-down menu of less common actions. After you complete this practical, take some time to explore the various action options. For now, you can close the AVD Manager window.

# TASK: Run your App

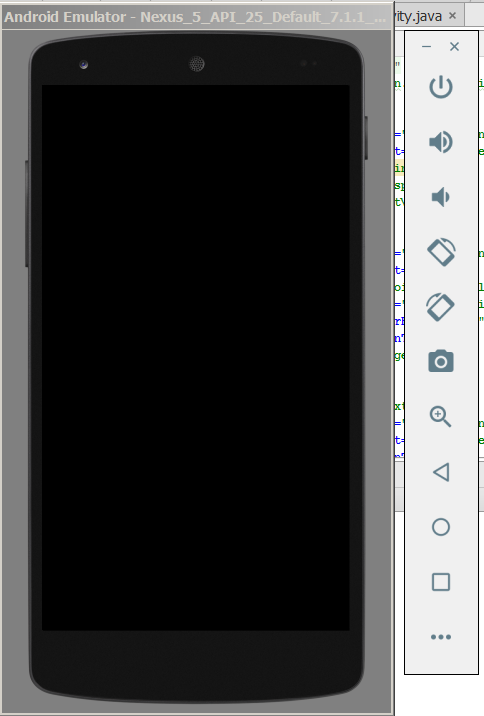
* To run your app, click the green triangle in the main AS icon ribbon (circled in the image below).



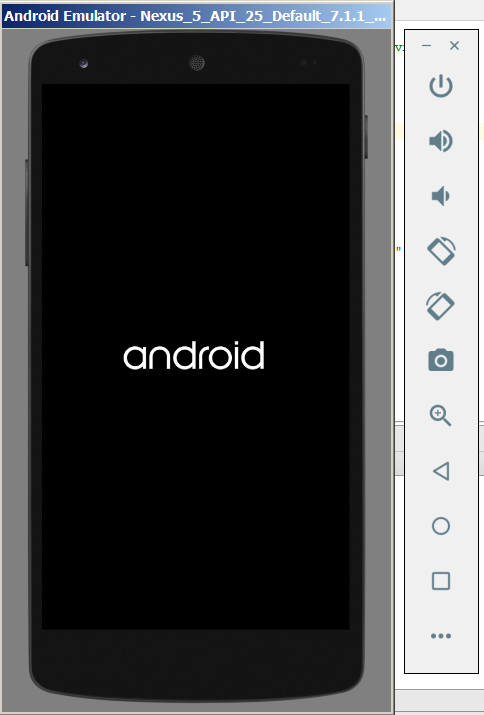
* You will be presented with the Device Chooser.



* Here you can either select an already running device (e.g. a previously launched emulator, or an attached physical phone or tablet), or you can launch one of your AVDs.
* Make your selection, then click Ok.
* Wait. Prepare to wait for a long time. The emulator will open in a separate window. For a while, it will look like this:



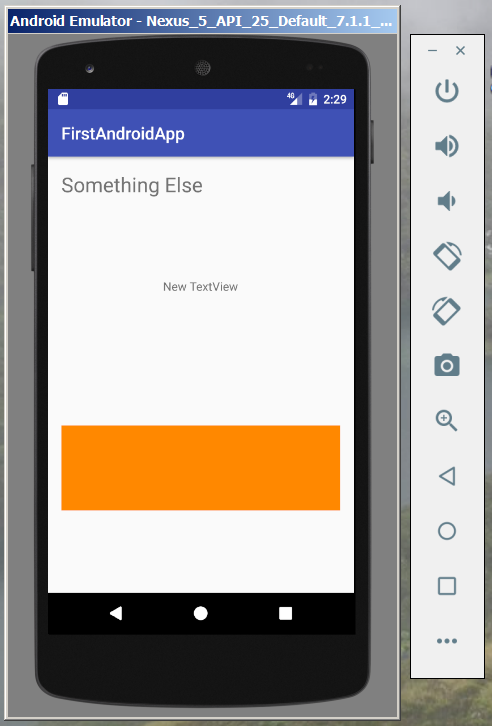
* Then for a while it will look like this…



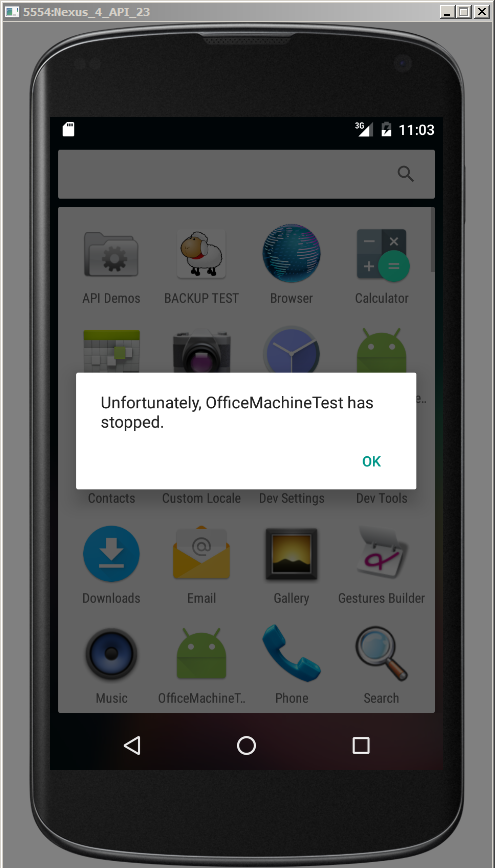
* At this stage, the emulator **IS NOT READY.** Do not click on it – it will break. Do not try to hurry it – there is nothing you can do. Just wait. Especially on a lower-powered machine, the emulator can take a long time to launch (several minutes is not impossible). You only have to go through this on the initial launch, not on subsequent runs of the application.
* When the emulator is fully launched, it will look like the start-up screen of a real phone. For example, the Nexus 5 looks like this:



* (Some AVDs will have little splash screens that you must click to close. Others will have little lock icons that you must “sweep” (click and drag the mouse). But until you see something that looks like a real phone’s start screen, the emulator is not fully launched.)
* When your emulator is fully launched, ***wait a little longer*** and your application should open automatically. It should look, in the emulator, much the same as it looks in Design view during development. If you had coded any interactive controls (we will start this next week), they would be live and responsive to mouse events.



* After your app opens, you can use the emulator’s “hardware” controls just like a real Android phone. The Home and Back buttons will background your app. The menu button will (assuming appropriate code has been written) raise menus. Your app will appear in the Apps menu and can be restarted by clicking on its icon. Etc.[[3]](#footnote-3)
* If your app has a bug that causes an exception to be thrown (e.g. divide by zero or null pointer), you will see the following (with the name of your app):



* This is the only error message you get from the emulator, and it is obviously not much help. Don’t worry -- Android Studio has excellent debugging tools, and we will learn to use them in the coming weeks.

1. From the Android documentation: *"The density-independent pixel is equivalent to one physical pixel on a 160 dpi screen...At run time, the platform transparently handles any scaling of the dp units needed, based on the actual density of the screen in use."* This insures that your layout will look correct on devices with different physical screen densities. [↑](#footnote-ref-1)
2. The AS built-in emulator is powerful, but requires a lot of hardware grunt to perform well. You will often see recommended an alternative tool called Genymotion which can be used with AS and tends to run faster. The OP student machines have Genymotion installed, but it is important to learn how to use the more powerful AS emulator, so we will use that one for this practical. [↑](#footnote-ref-2)
3. If you are not accustomed to using an Android phone, take some time to experiment with the interface, and/or ask for help at this point. [↑](#footnote-ref-3)