Intro to Android Development - Part 1

The Android mobile operating system is the leading smartphone and tablet operating system. From a minority operating system only running on one phone (the T-Mobile G1) a few years ago it has grown into one of the leaders in the smartphone operating system field, along with iOS (on the iPhone), and to some extent, Windows Mobile.

Why develop for Android?

There is one very important difference between Android and many of its competitors. It is an open source operating system, which means that you are free to modify it for your own ends. Even more crucially for app developers is that as a result of its open-source nature ***you can develop and distribute applications without restriction***. This is in contrast to some of the other contemporary mobile development options in which the operating system vendor restricts distribution to a single channel owned by themselves, and "vets" software before making it available. Android has an official distribution channel (the Play Store), but this is more liberal with accepting apps than some of the other vendors.

Android versions

At this point it is worth elaborating on the various ***versions of Android***. At the time of writing (see the stats on android.com), the most recent versions deployed on actual mobile devices are 9.0 (Pie); 8.0 and 8.1 (Oreo); however many devices are running 4.4 (KitKat), 5.0 and 5.1 (Lollipop); 6.0 (Marshmallow) and 7.0 and 7.1 (Nougat). Marshmallow is currently the most installed version (16.9% as at early May 2019).

It can be seen from the stats above that significant numbers of devices are still running Lollipop (5.x) and around 7% version 4.4 (KitKat); therefore, targeting 4.4 upwards, or 5.0 upwards, is a sensible approach. The ***support library*** (which will be covered later) makes this easier.

Another concept that you need to understand is the ***API level***. The ***Android API*** is the set of Java classes which are used to program Android apps with. The API level denotes revisions to the Android API, in a sequence of positive integers starting from 1. Thus, the numbers used for the API levels are not the same as those used for the Android versions, but each API level corresponds to a particular version. The idea is that each time Android itself is updated, the API is updated too. For example:

- API level 15 corresponds to Android 4.0.3;
- API level 19 corresponds to Android 4.4;
- API level 23 corresponds to Android 6;
- API level 26 corresponds to Android 8.0;
- API level 28 corresponds to Android 9

When developing an Android app, you have to specify the minimum API level on which your app will run. Thus an app with minimum API level 19, for example, will only run on 4.4+. As seen above, the vast majority of devices are running at least API level 19 (Android 4.4; KitKat), so if you specify API level 19 as a minimum you will be targeting the majority of devices.

Runtime Environment: ART and Dalvik

With standard Java, you compile to bytecode which is then run using the Java Virtual Machine (JVM). Android is similar but rather than using the standard JVM, it uses its own virtual machine and corresponding bytecode format (DEX format). So "regular" Java bytecode will not run on Android and Android apps will not run on a regular JVM.

Dalvik was the original virtual machine, which versions of Android up to 4.4 used. With Android 5.0, a new virtual machine (***ART : Android Runtime***) (see here) is used instead. See here for more details on Android virtual machines.

General nature of Android development

Android development is generally done in the Java programming language. However, because the environment differs from a standard desktop PC, the actual libraries available differ somewhat from the standard Sun/Oracle Java Development Kit. Many standard Java features from packages such as java.io and java.util are available; however (as you might expect) the standard Java GUI libraries (designed for desktop applications) are not, and also the structure of an Android application is significantly different due to the different style of interaction with a mobile device compared to a desktop computer.

What do you need to start Android development?

To get started on Android development you ideally need the ***Android Studio*** IDE. Android Studio is Google's custom IDE for Android development; it is now the official IDE for Android.

It is also possible to develop apps purely using command-line (console) tools but we will not cover this in this unit.

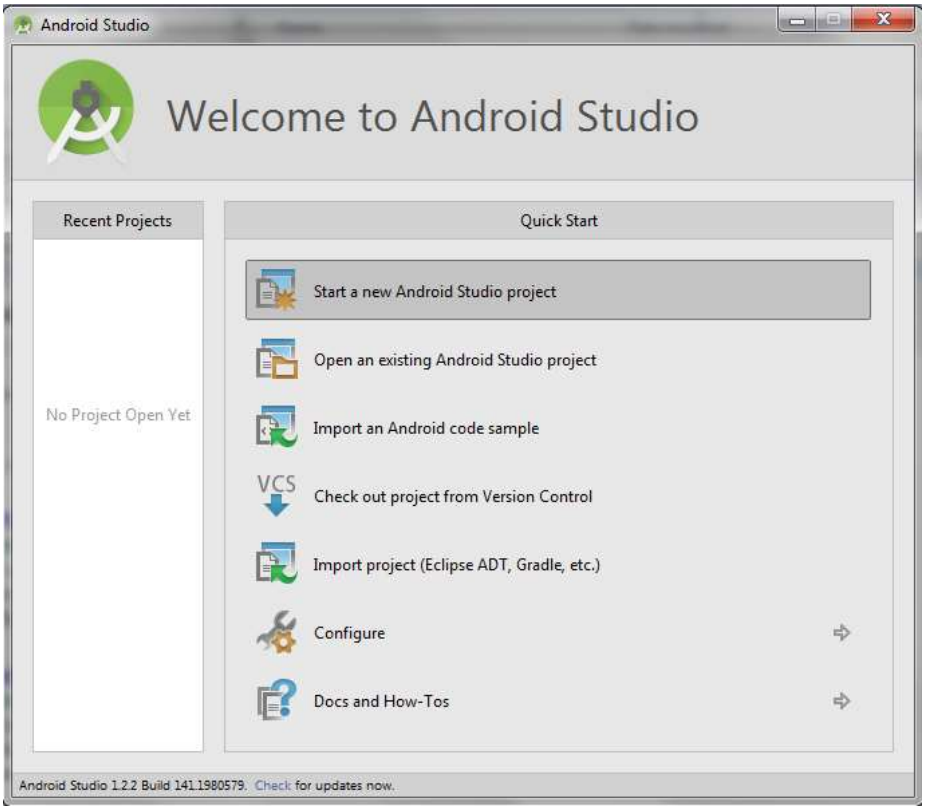
Android Studio provides an IDE 'wrapper' round the core Android development environment, which contains the following components:

- The Android Software Development Kit (SDK): provides Android libraries, tools to generate bytecode and distributable apps and the ***SDK Manager*** which allows you to download libraries for different versions of Android;
- The ***AVD (Android Virtual Device) Manager***: allows creation of virtual phones or tablets allowing you to test your app in the absence of a real device.

You can access the Android SDK and the AVD Manager either through an IDE such as Android Studio or independently, via the command-line.

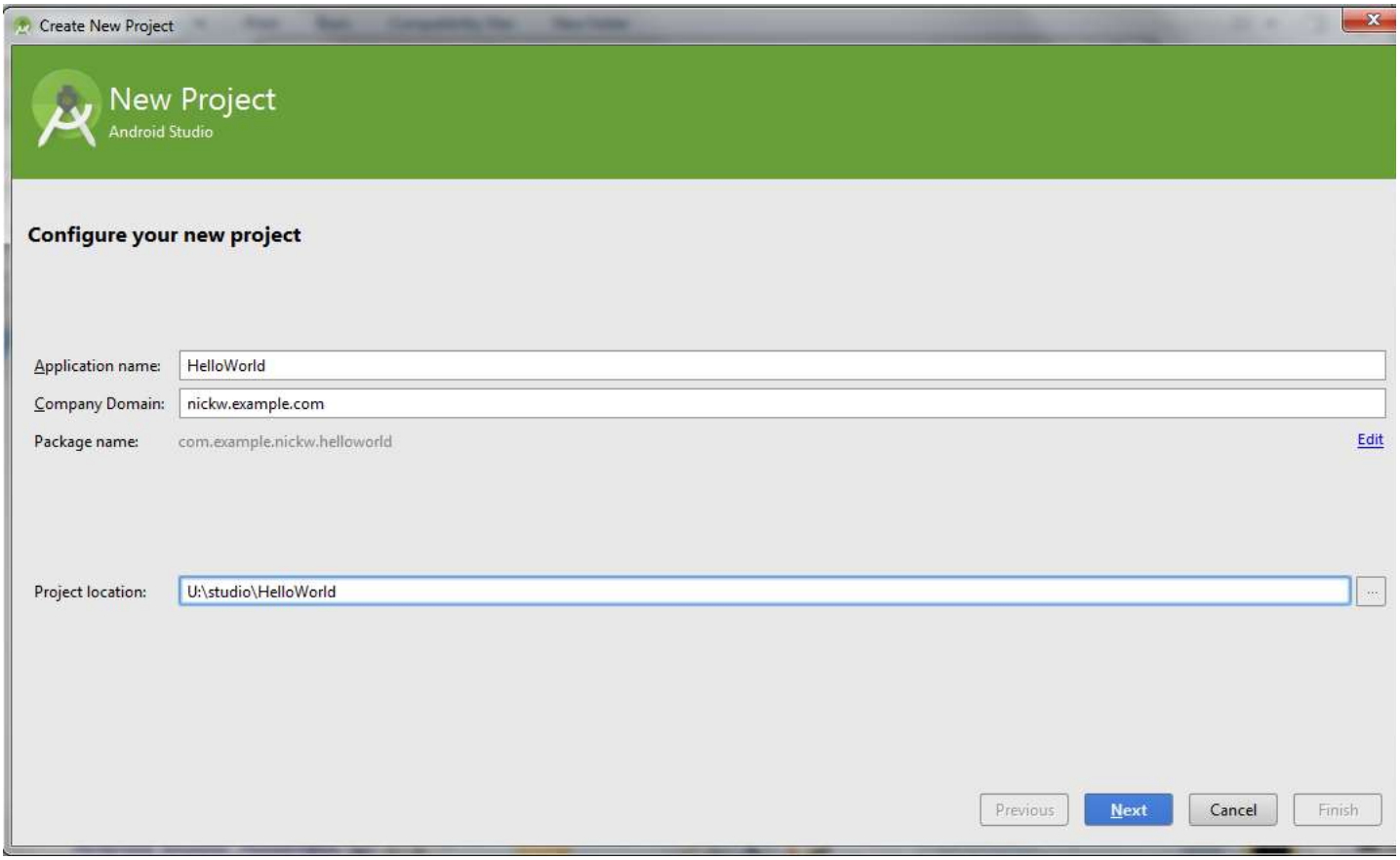
Getting started

The best way to explore the different components of the Android development environment is to get started with Android Studio. Launch Studio, you will see a screen like this:



Creating a project

Select "Start a new Android Studio project". You will then see this screen. This allows you to create a new project. ***See the important points below on where you should save your project.***

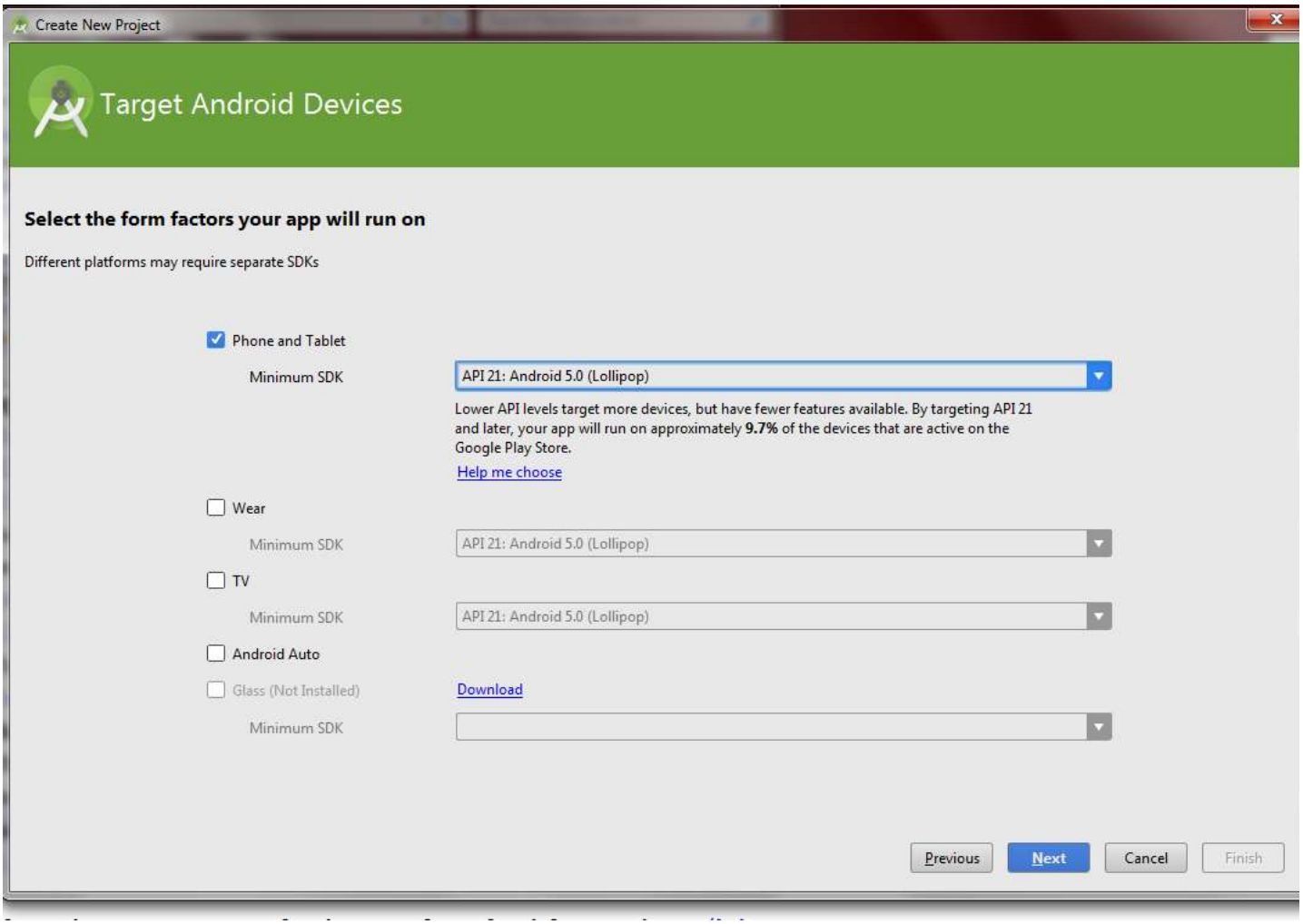


To explain these one by one:

- ***Application name***: the name of your application.
- ***Company domain***: a unique identifier for yourself or your organisation. Typically this will be your domain name (if you have one) but ***example.com*** can be used as a placeholder, hence ***nickw.example.com***. Obviously replace ***nickw*** with your name!
- ***Project location***: where your projects will reside. You should make sure that this is ***on the C: drive*** (under ***C:\Users\your username***). (Note that the U: drive or USB stick will be too slow).
  HOWEVER, you should then ***copy it to the U: drive or USB stick, or push to GitHub*** at the end of the class as the C: drive will be ***wiped*** when you logout! Make a folder called ***studio*** within ***C:\Users\your username*** to store your projects. Each project will automatically be stored in a subfolder with the same name as the project name. For example, ***C:\Users\1smitj01\studio\HelloWorld***.

Choosing a target device and Android version

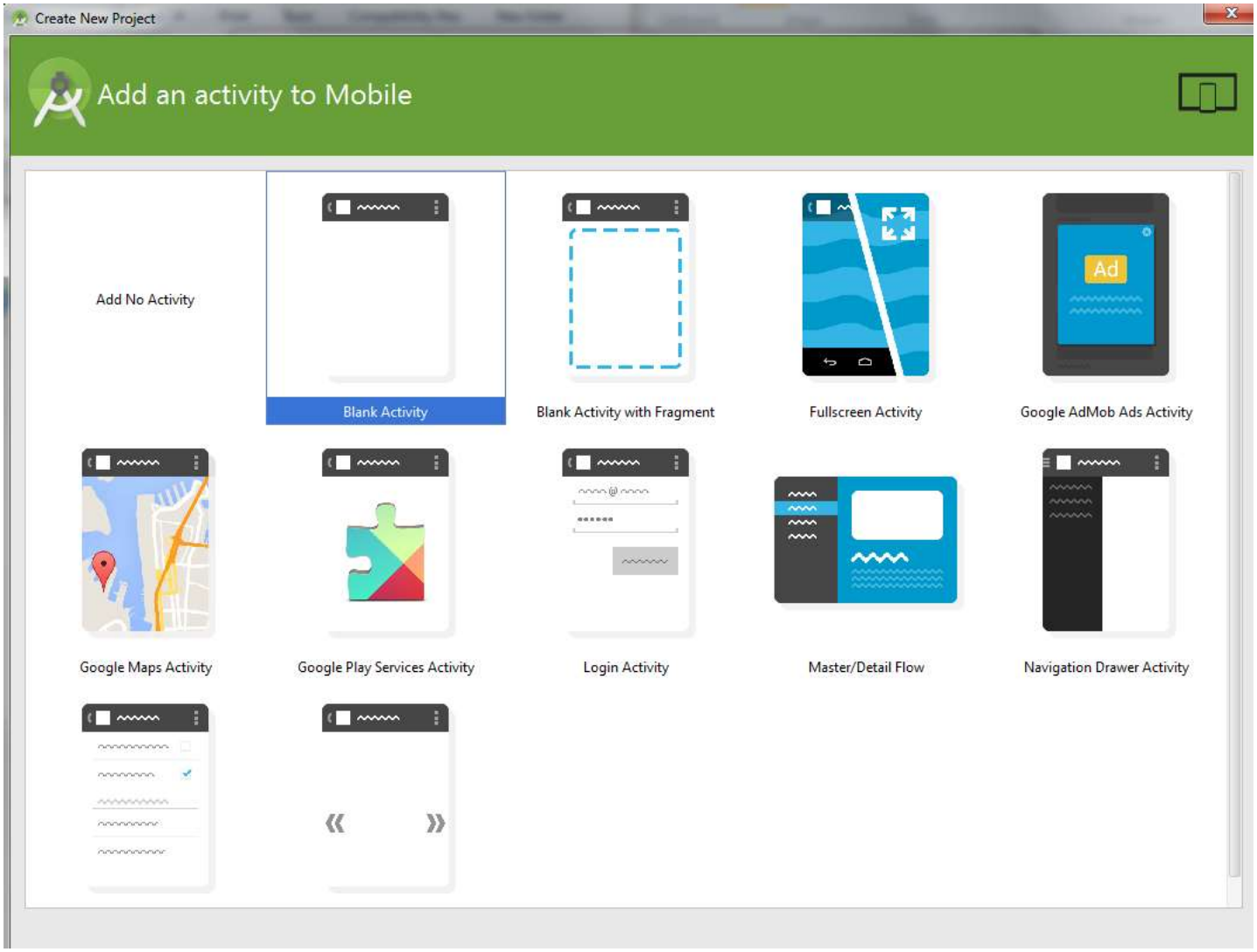
Once you have done this, the following screen will appear:



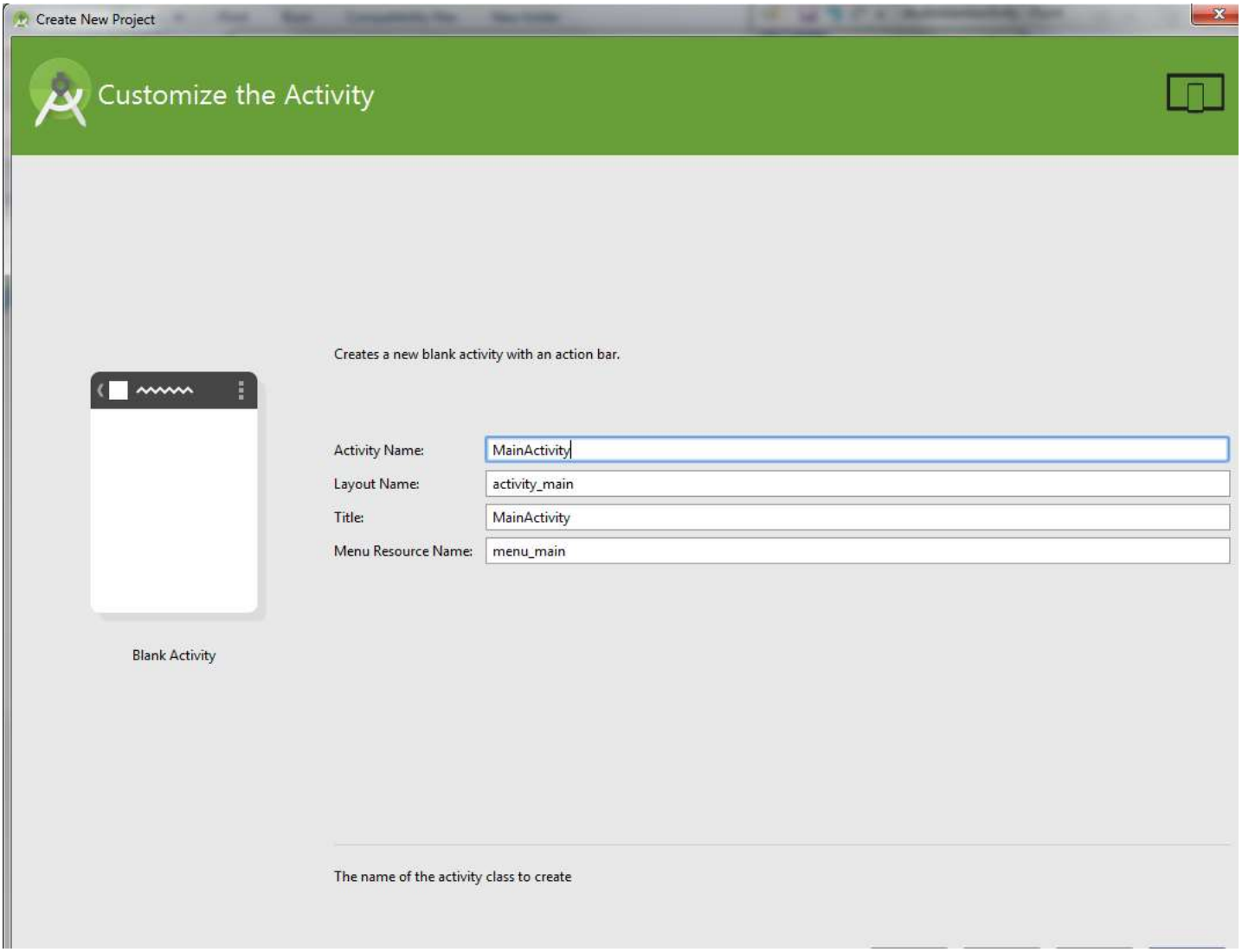
This screen allows you to specify the target devices for your app. It allows you to target standard mobile devices (phones and tablets) or more specialised devices such as smartwatches. It also allows you to specify the minimum version of Android that your app supports. Choose 4.4 (API 19; KitKat)

Adding an activity

The next screen allows you to add an ***activity*** to your project. An ***activity*** is an an individual screen of your app. Apps typically have several activities with a ***main activity*** which represents the main screen of your app, which appears when you first launch it.

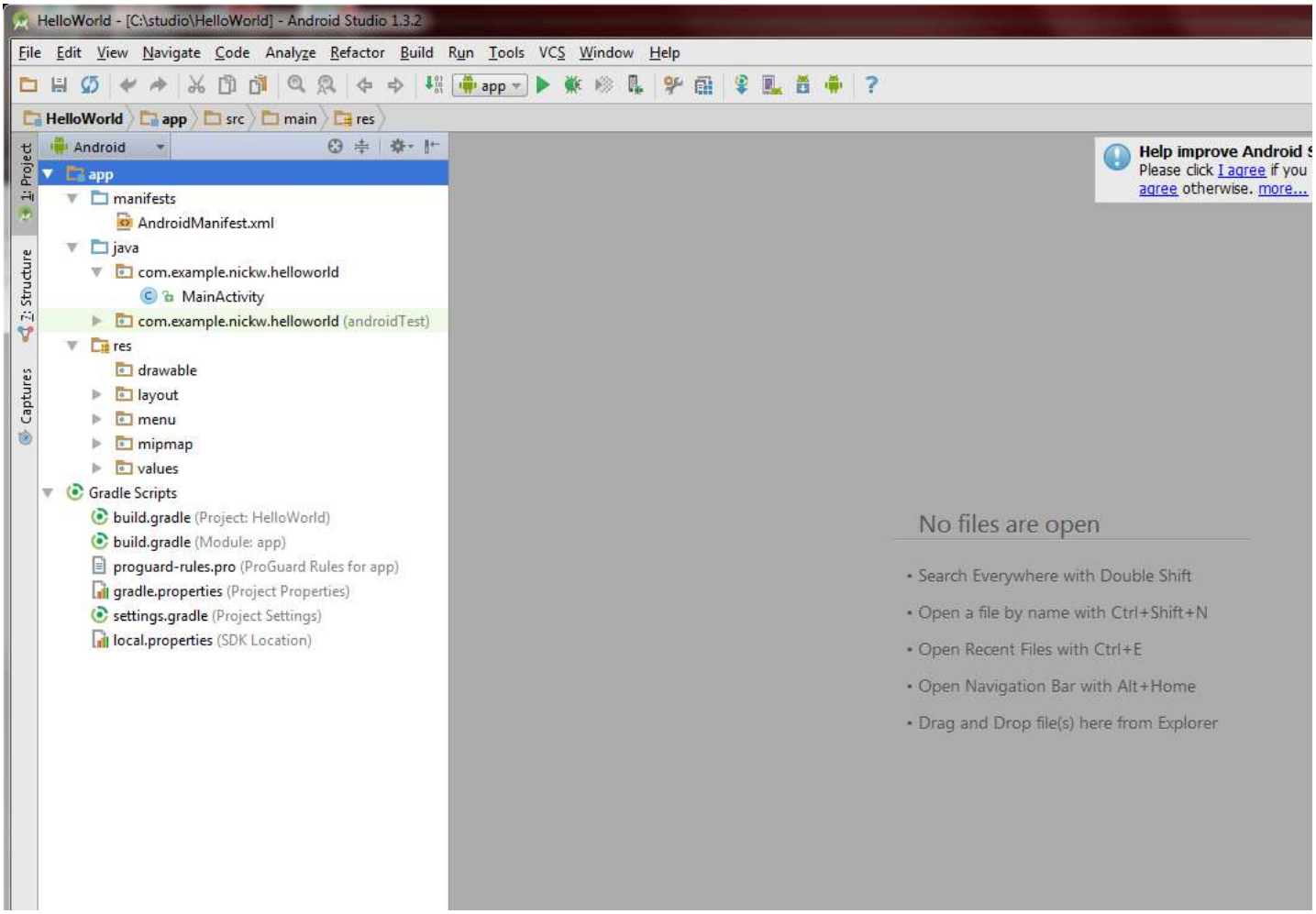


Choose the "blank activity". This will create an activity with little or no code added to it. In the next screen, you need to specify the name of the activity, just enter "MainActivity" as in the screenshot below. The other entries (which will be expanded upon later) will be automatically filled in.



Explaining the layout of a project

If you expand the project layout on the left the layout will look something like this:



To explain each entry:

- ***manifests***: this contains the ***manifest file***, AndroidManifest.xml, which contains information about the app and its contents. We will return to this later.
- ***java***: the actual code. Note how it contains our activity, ***MainActivity.java***
- ***res***: the ***resources***. These are additional files our app needs, such as screen layouts, menus, images and so on. We will examine resources in more detail later.
- ***Gradle Scripts***: Gradle is a tool to automate the process of building an application. It links in third-party Java libraries (***dependencies***) our app needs to run, downloading them from online repositories if necessary. The key Gradle script is ***build.gradle***. In this file, we also specify the minimum Android version supported by our app. Note that the important ***build.gradle*** is the one in the ***app*** folder, not the other one.

Hello World!

We're now going to start - as always in software development - with a Hello World app. You should see pre-generated code something like this:

```
package whatever.your.package.is; // leave this line as it was
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;

public class MainActivity extends AppCompatActivity {

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

What does this code mean?

- As mentioned above, the entry point to an Android application is an ***Activity***. An Activity is basically a single screen of an Android app. It contains "widgets" or user interface elements, through which the user can interact with the app: for example, text fields, buttons and radio buttons. An Android app will typically consist of several Activities, representing different screens within the app, but there will be one main activity which launches when the app is launched. The main activity is defined in the manifest file (see later)
- The ***onCreate()*** method is the actual entry point to the activity. It can be viewed as roughly equivalent to the ***main()*** method in standard Java, or in C or C++. Whenever an activity is created for the first time, its ***onCreate()*** method is called: therefore, initialisation code should be placed in onCreate().
- Note that the activity here is an ***AppCompatActivity***, which is a subclass of plain ***Activity***. This allows us to use the support library to include newer Android API features on older versions of Android. We will return to this later.
- The first thing we do is call the version of ***onCreate()*** in the superclass (i.e. AppCompatActivity). In this way, we can ensure that common functionality which occurs when all activities are created will also occur in our case.
- We then set up the layout. Don't worry about what the ***R*** means just yet: instead, take out the line

```
setContentView(R.layout.activity_main);
```

and replace it with

```
TextView tv=new TextView(this);
tv.setText("Hello World!");
setContentView(tv);
```

You will also need to add

```
import android.widget.TextView;
```

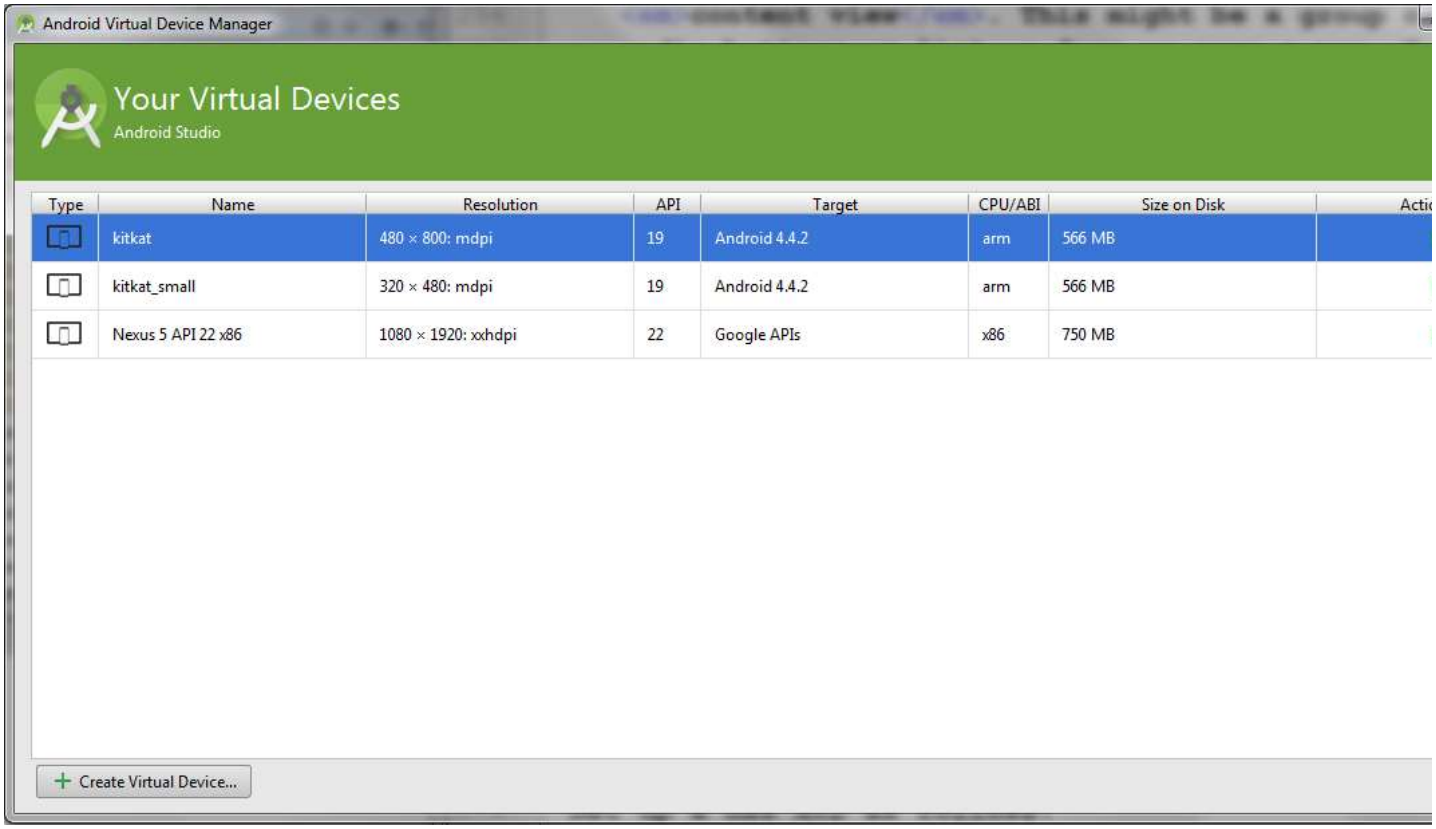
in the imports section at the top of the file.

- Hopefully this code is fairly obvious but it does illustrate a couple of key concepts. The first is the concept of a ***View***. Every Activity contains a View which is the "main component" of the screen, referred to as the ***content view***. This might be a group of radio buttons, a list, a form or even a map. Here, it's the simplest type of View: a ***TextView***. A TextView is what it sounds like: a View which can contain text. So here, we create a TextView containing the text "Hello world" and make it the content view of our Activity.

Setting up an Android Virtual Device

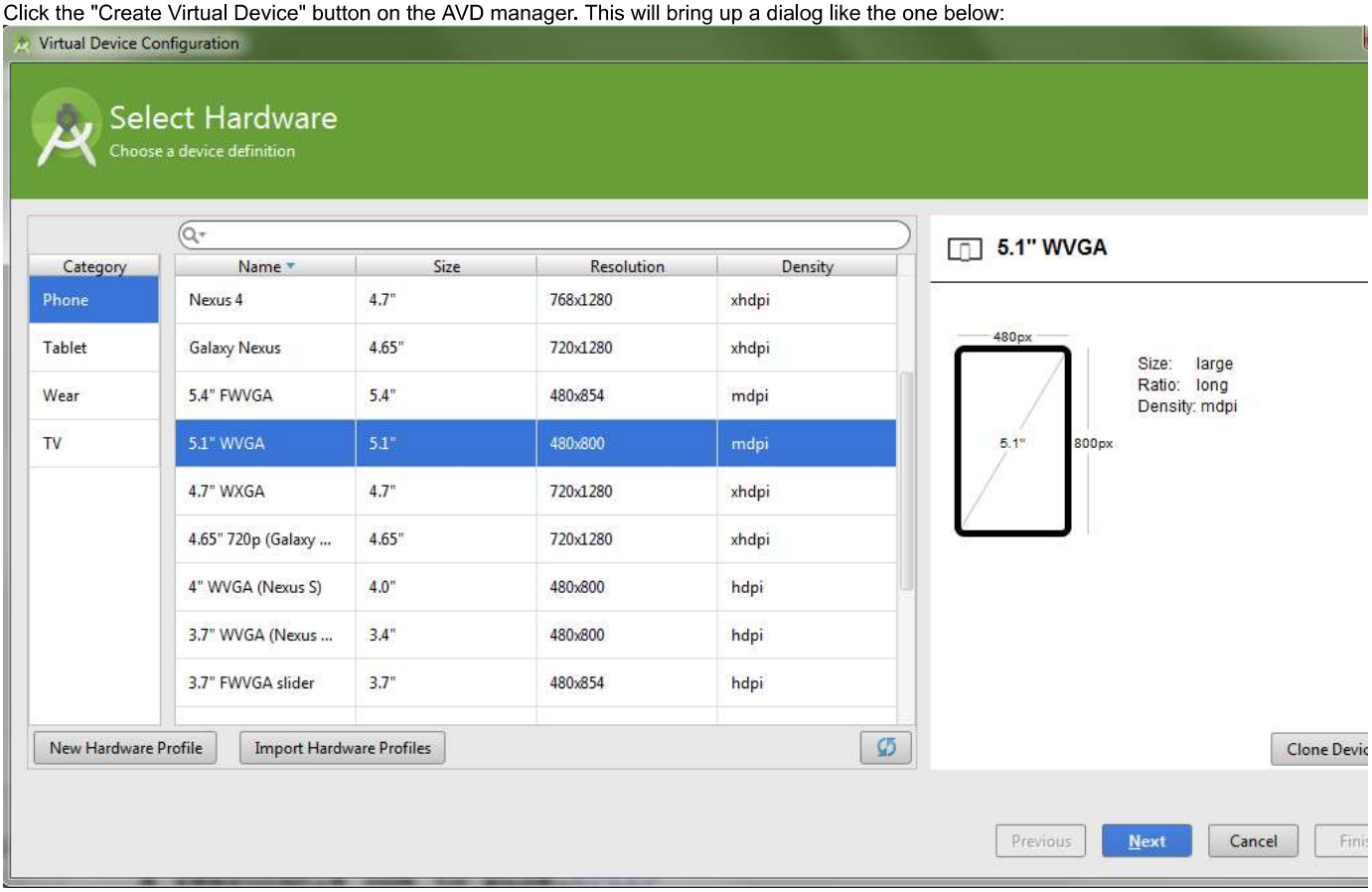
If you do not have an actual Android device, you will need to set up an ***Android virtual device (AVD)*** before starting programming. This is an emulator which you can use to test your apps as you develop them. It resembles an actual phone, and the user interface looks just like a user interface on a real Android phone so you can test your apps fairly realistically. When creating an AVD you will be prompted for various properties of the emulator such as resolution.

To set up an AVD in Studio, select ***Tools-Android-AVD Manager***. This will launch the Android AVD manager from within Android Studio, as shown below:

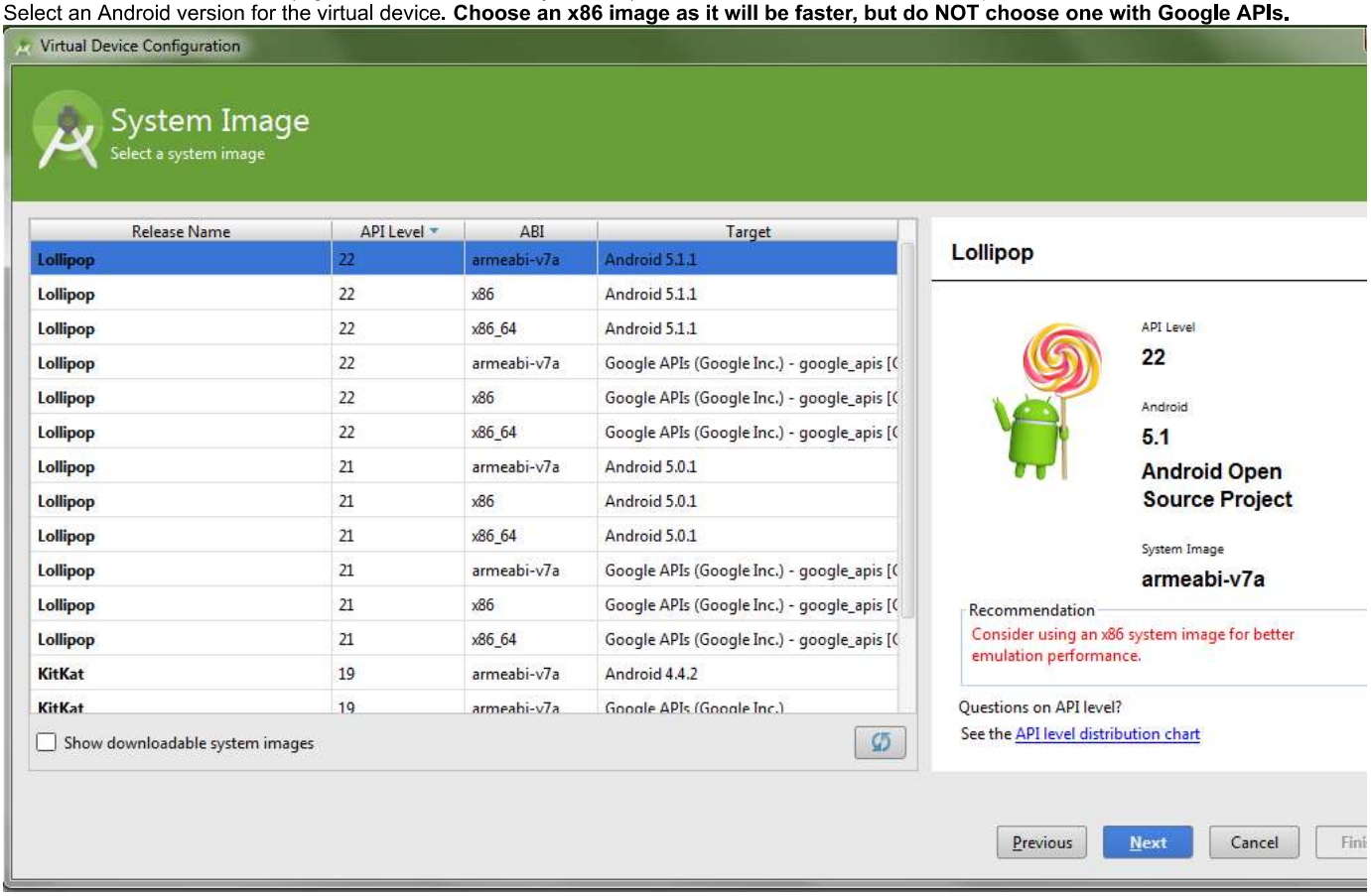


This screen shows that there are three AVDs already set up, called ***kitkat***, ***kitkat_small*** and ***Nexus 5 API 22 x86***. On your system there probably won't be any yet. So Set up a new AVD as follows:

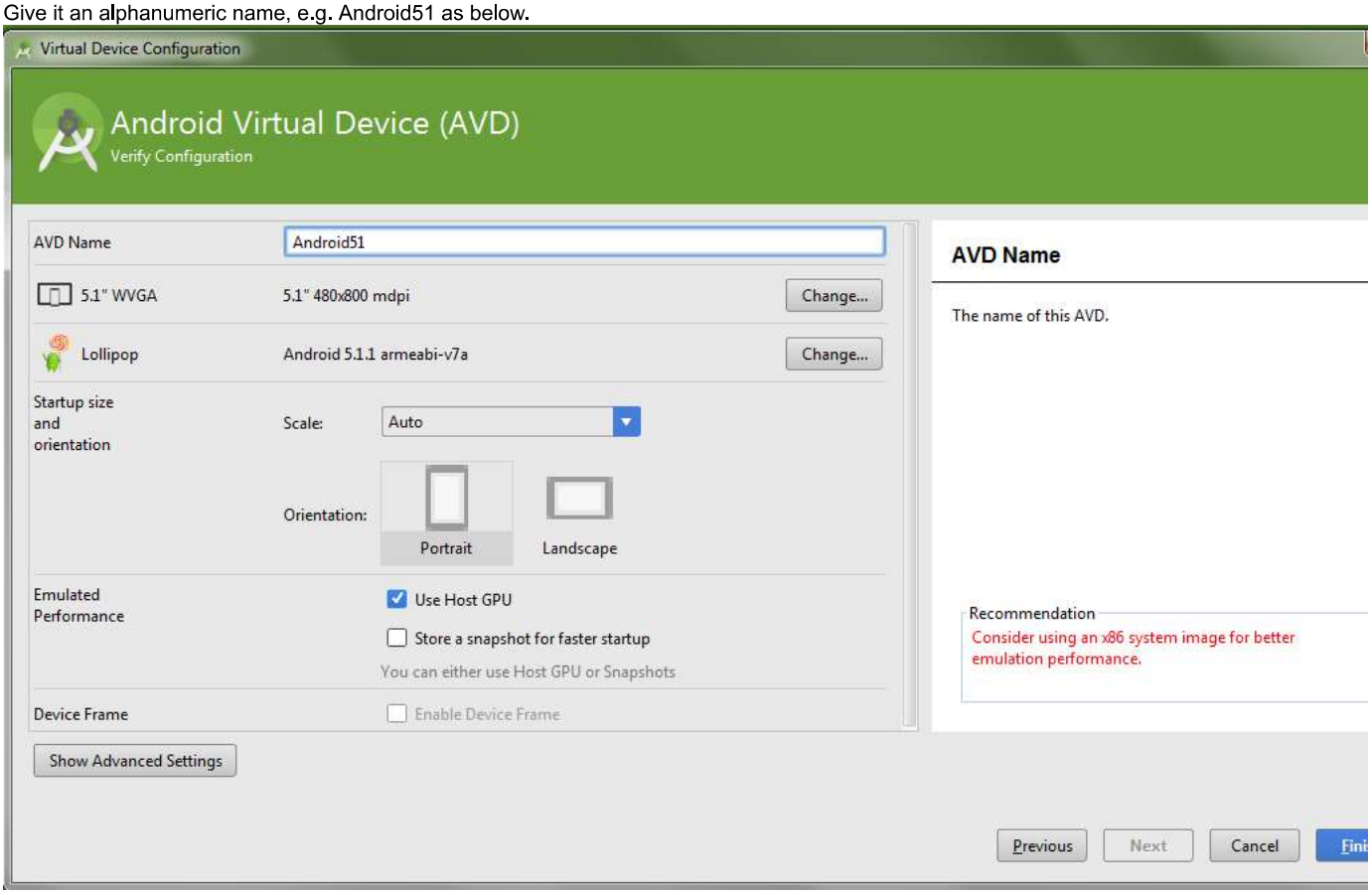
- Click the "Create Virtual Device" button on the AVD manager. This will bring up a dialog like the one below:



- Select one with low resolution (higher-resolution ones may be slow). ***5.1" WVGA*** is a reasonable one to pick.
- Select an Android version for the virtual device. **Choose an x86 image as it will be faster, but do NOT choose one with Google APIs.**



- Give it an alphanumeric name, e.g. Android51 as below.



- Once setup, your new AVD will appear on the "Your Virtual Devices" screen.

Running on an actual device

You can test your apps on an actual device, though you have to enable the developer settings. Full instructions on this are available from the Android site. On Linux and Mac OS X as no driver is required. On Windows, a driver for your device is required, however Nexus devices can use the Google USB driver which comes with the SDK. For other devices, you can download a driver for your device from the Android developer site. See the Android documentation on driver installation for more details. To summarise, you have to ***download*** and then ***install*** the driver. Even with the Google USB driver, included in the SDK, the ***installation*** step is necessary.

The Google USB Driver has been installed in the labs which means that you can use Nexus and other Google-branded devices (e.g. Pixel) to test your apps on. The Samsung driver should also have been installed, meaning Samsung phones should also work. If you have another Android device, you will need to use the emulator.

The Android SDK Manager

One of the most crucial components of the Android SDK is the ***SDK Manager***. This piece of software, which can be run within Android Studio or standalone, allows you to download and install versions of the SDK for different versions of Android, along with other items such as documentation. So if a new version of the SDK is released, the SDK manager allows you to download that new version. The default Studio download only comes with the latest versions of the SDK, so if you want to target older devices, you need to download older versions.

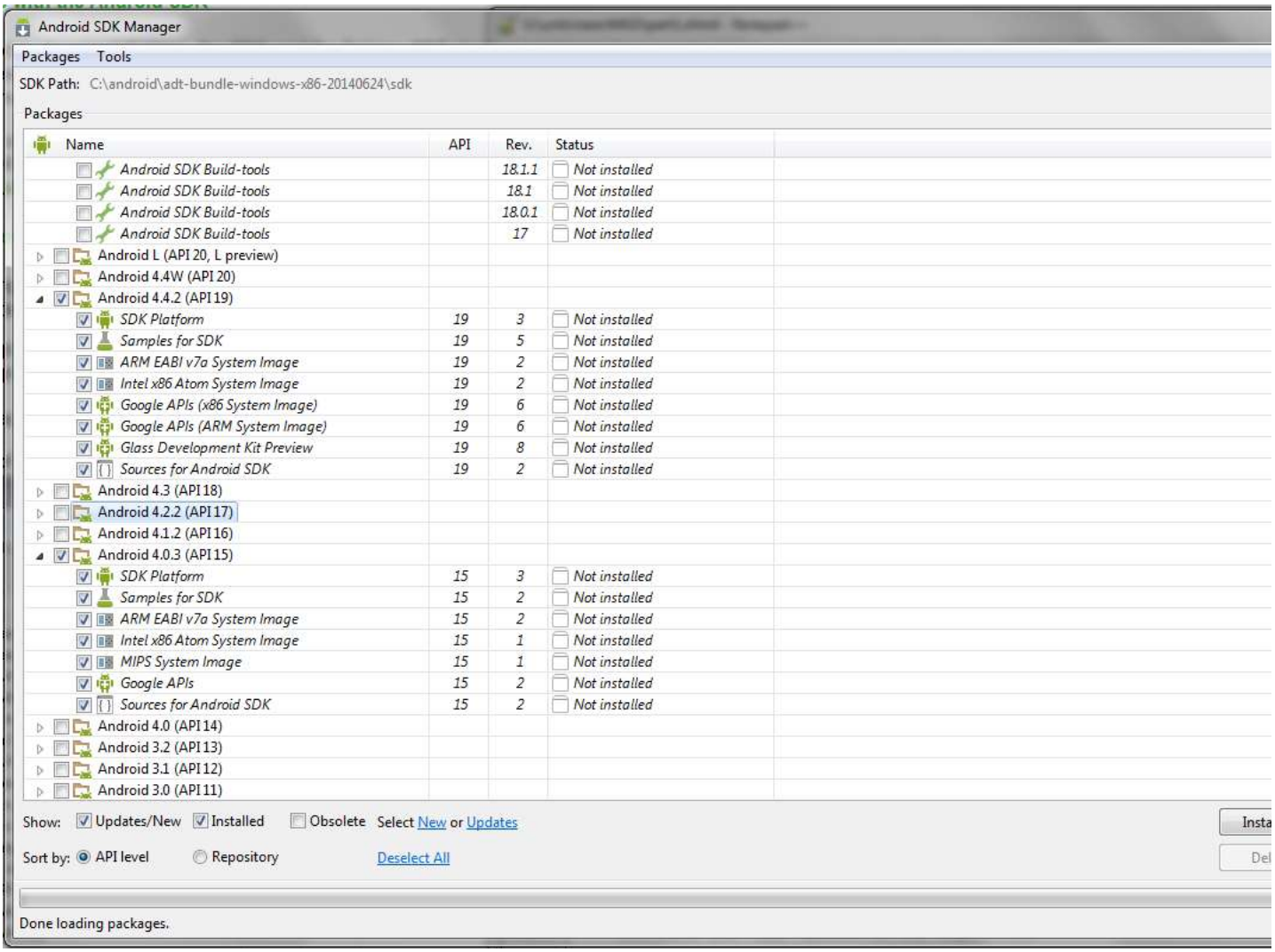
Starting the SDK Manager

On Android Studio, select ***Tools-Android-SDK Manager***. This will launch an intermediate screen listing installation options; however for more control it's recommended you then click on ***Launch Standalone SDK Manager*** which will give you the SDK Manager as it appears if you launch it on its own without Android Studio.

Installing SDK versions from the Android SDK manager

You might want to install older SDK versions than those included by default by Android Studio. For example, you might want to install Android 4.4.2 and 4.0.3 (API levels 19 and 15 respectively). Start the SDK manager as described above and select these two versions, as shown in the screenshot below

- Android 4.4.2 (API 19);
- Android 4.0.3 (API 15).



You will then need to accept the licence and it will download the individual components of these versions of the SDK.

XML Layouts

The Hello World example above will probably look familiar if you have done any programming in desktop GUI toolkits, such as Java AWT or Swing. We add components to the main screen programatically. However with Android we can cut down on the amount of setup code using ***XML layouts*** instead. With XML layouts, we define the layout of the content view of the Activity using XML tags; in that respect, there is some similarity with writing an HTML web page, but whereas websites are driven by a browser's engine, with apps ***we*** are writing the engine and are thus much more in control. (In case you are not aware, XML is a tag based format for representing data). XML layouts have a number of advantages, they include:

- Separating the layout keeps the Java focused on the app's logic, meaning that it's easier to understand the "nitty-gritty" of the app without having to wade through GUI setup code;
- We can change the layout without recompiling, simply by changing the XML; the XML is loaded by the application when it runs;
- Separation of the work of designers and developers: designers can work on the app's layout by manipulating the XML code without needing to know any Java.

Here is an example of an XML based layout:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >
<TextView android:layout_width="wrap_content"
android:layout_height="wrap_content" android:text="@string/hello" />
</LinearLayout>
```

To explain this:

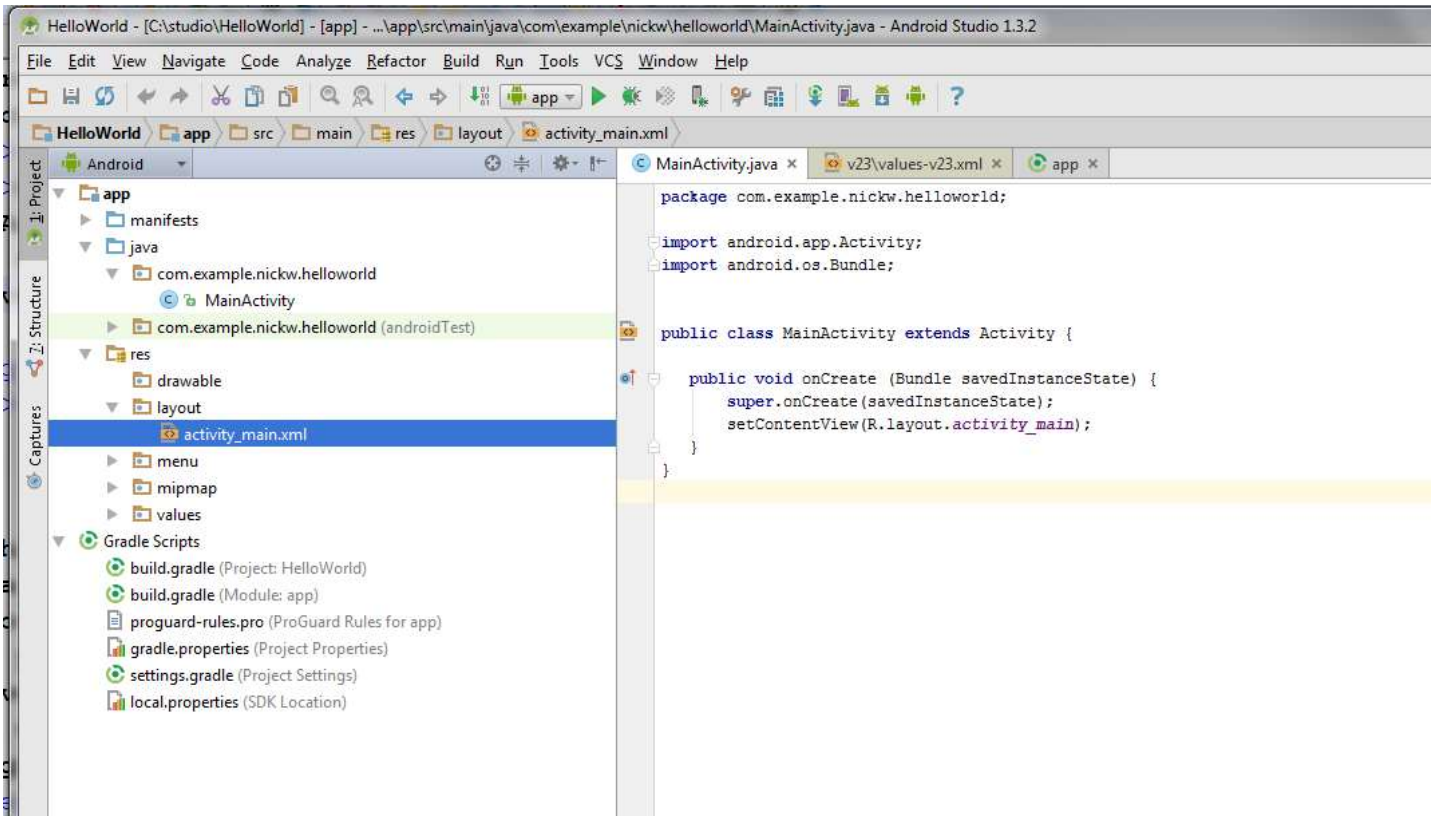
- The ***LinearLayout*** defines the view's layout. The layout specifies how the various components are arranged with respect to each other. LinearLayout indicated that components are placed adjacent to each other, either vertically or horizontally (depending on the ***android:orientation*** attribute)
- We then add a TextView tag. Basically every Android UI class has corresponding tag of the same name.
- The ***android:layout_width*** and ***android:layout_height*** attributes of the ***TextView*** describe how it fills its parent layout in the horizontal and vertical directions. The most common values are ***match_parent*** and ***wrap_content***. The former, as used here, means that the TextView entirely fills its parent (the LinearLayout), the result being that the TextView fills the entire screen. The latter,***wrap_content***, means that ***enough space to contain the content*** (the text "Hello World" here) should be used.

- The ***android:text*** attribute specifies the text within the TextView. Note however how we don't give it a straight value, we use ***@string/hello***. This i described in more detail below.

Where can the XML file be found? It is found in the ***application resources***, described below.

Application resources

Android apps consist of Java code plus ***resources*** - additional data which the app needs to do its job. An example of a resource is an XML layout file, as described above. If you expand the ***res*** folder in Android Studio, you will see the layout below.



Within the ***res*** folder is a ***layout*** folder, and within that is the ***activity_main.xml*** which is auto-generated when you start a new Android project: it is basically a default main view for your main Activity. You replaced it with code to manually create a TextView object above. However, now you are going to change it back to reading the layout from the XML file. Replace (or comment out) the code you wrote above with this code (which was in the original auto-generated code for your main Activity):

```
setContentView(R.layout.activity_main);
```

Run your app again and you will see that this message now appears:

```
Hello World!
```

Where is this message coming from? Remember that in the ***activity_main.xml*** file, the ***android:text*** attribute of the TextView describes the text on the TextView and this is set to the value ***@string/hello***. What does ***@string/hello*** represent? It is a ***string resource***. In Android development, to make it easier to translate apps into different languages, much of the text that we see within the user interface is defined in a ***string resource file*** so that we can easily translate an app to a different language simply by editing the string resource file. This can be found within the ***values*** folder within ***res***, in the file ***strings.xml***. If you look in the ***strings.xml*** file, you will see this line:

```
<string name="hello">Hello World!</string>
```

Change this to:

```
<string name="hello">Geia sas Kosmos!</string>
```

(Sorry if I have got the Greek wrong!). Run your app again, and you should find it greets you with ***Geia sas Kosmos!*** Notice how each string in the ***strings.xml*** file has a ***name***, and we reference that name in the layout file with ***android:text="@string/(name of string)"***.

As well as layout and strings, the ***res*** folder can contain other types of resource. These include application menus (which we will come onto a bit later on) and images. When you distribute an app, all the resources are packed into one file along with your actual code.

The auto-generated R.java file

You might be a bit puzzled as to the meaning of the ***R*** in the code you added above, i.e.

```
setContentView(R.layout.main);
```

What, actually, is this "R"? It's a pre-generated class which contains "hooks" into your XML resource files. The ***R.java*** file, not directly accessible from Studio but present in your project, looks something like this:

```
/* AUTO-GENERATED FILE.  DO NOT MODIFY.
 *
 * This class was automatically generated by the
 * aapt tool from the resource data it found.  It
 * should not be modified by hand.
 */

package com.example.nickw.helloworld;

public final class R {
    public static final class attr {
    }
    public static final class drawable {
        public static final int icon=0x7f020000;
    }
    public static final class layout {
        public static final int activity_main=0x7f030000;
    }
    public static final class string {
        public static final int app_name=0x7f040001;
        public static final int hello=0x7f040000;
    }
}
```

Notice that it contains several static variables. These are identifiers which your Java code can use to access the XML, for example ***R.layout.activity_main*** has the hex value ***0x7f030000*** which is a "handle" for the activity_main.xml resource file. Every time you add a resource to an Android app, your ***R.java*** will automatically be updated and you will be able to use static attributes of ***R*** in your Java code to access different resources. Never edit R.java directly by the way, the system will always do it for you!

Further reading

You might want to check out these resources for further reading:

- The official Android developers' site
- An in-depth tutorial on Android development