

Chapter 1: Getting started with Android

Version	API Level	Version Code	Release Date
1.0	1	BASE	2008-09-23
1.1	2	BASE 1 1	2009-02-09
1.5	3	CUPCAKE	2009-04-27
1.6	4	DONUT	2009-09-15
2.0	5	ECLAIR	2009-10-26
2.0.1	6	ECLAIR 0 1	2009-12-03
2.1.x	7	ECLAIR_MR1	2010-01-12
2.2.x	8	FROYO	2010-05-20
2.3	9	GINGERBREAD	2010-12-06
2.3.3	10	GINGERBREAD_MR1	2011-02-09
3.0.x	11	HONEYCOMB	2011-02-22
3.1.x	12	HONEYCOMB_MR1	2011-05-10
3.2.x	13	HONEYCOMB_MR2	2011-07-15
4.0	14	ICE_CREAM_SANDWICH	2011-10-18
4.0.3	15	ICE_CREAM_SANDWICH_MR1	2011-12-16
4.1	16	JELLY_BEAN	2012-07-09
4.2	17	JELLY_BEAN_MR1	2012-11-13
4.3	18	JELLY_BEAN_MR2	2013-07-24
4.4	19	KITKAT	2013-10-31
4.4W	20	KITKAT_WATCH	2014-06-25
5.0	21	LOLLIPOP	2014-11-12
5.1	22	LOLLIPOP_MR1	2015-03-09
6.0	23	M(Marshmallow)	2015-10-05
7.0	24	N(Nougat)	2016-08-22
7.1	25	N_MR1 (Nougat MR1)	2016-10-04
8.0	26	O (Developer Preview 4)	2017-07-24

Section 1.1: Creating a New Project

Set up Android Studio

Start by setting up Android Studio and then open it. Now, you're ready to make your first Android App!

Note: this guide is based on Android Studio 2.2, but the process on other versions is mainly the same.

Configure Your Project

Basic Configuration

You can start a new project in two ways:

- Click Start a **New** Android Studio Project from the welcome screen.
- Navigate to **File** → **New** Project if you already have a project open.

Next, you need to describe your application by filling out some fields:

1. **Application Name** - This name will be shown to the user.

Example: `Hello World`. You can always change it later in `AndroidManifest.xml` file.

2. **Company Domain** - This is the qualifier for your project's package name.

Example: `stackoverflow.com`.

3. **Package Name** (aka `applicationId`) - This is the *fully qualified* project package name.


It should follow *Reverse Domain Name Notation* (aka *Reverse DNS*): *Top Level Domain . Company Domain . [Company Segment .] Application Name*.

Example: `com.stackoverflow.android.helloworld` or `com.stackoverflow.helloworld`. You can always change your *applicationId* by overriding it in your gradle file.

Don't use the default prefix "com.example" unless you don't intend to submit your application to the Google Play Store. The package name will be your unique **applicationId** in Google Play.

4. **Project Location** - This is the directory where your project will be stored.

Create New Project



New Project

Android Studio

Configure your new project

Application name:

Company Domain:

Package name: [Edit](#)

Project location:

Select Form Factors and API Level

The next window lets you select the form factors supported by your app, such as phone, tablet, TV, Wear, and Google Glass. The selected form factors become the app modules within the project. For each form factor, you can also select the API Level for that app. To get more information, click **Help me choose**

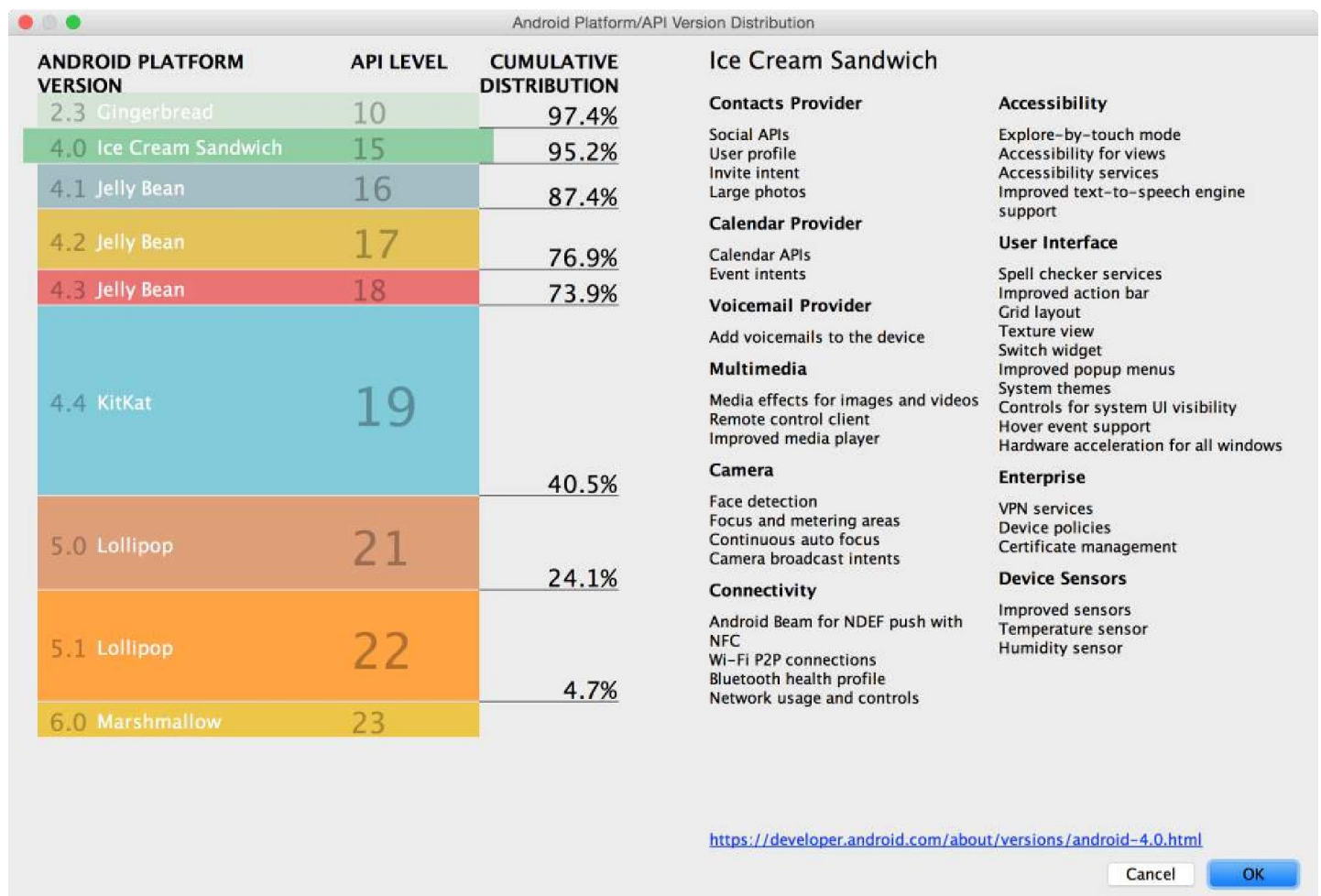



Chart of the current Android version distributions, shown when you click Help me choose.

The Android Platform Distribution window shows the distribution of mobile devices running each version of Android, as shown in Figure 2. Click on an API level to see a list of features introduced in the corresponding version of Android. This helps you choose the minimum API Level that has all the features that your apps needs, so you can reach as many devices as possible. Then click **OK**.

Now, choose what platforms and version of Android SDK the application will support.

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)

Cancel

Previous

Next

Finish

For now, select only *Phone and Tablet*.

The **Minimum SDK** is the lower bound for your app. It is one of the signals the Google Play Store uses to determine which devices an app can be installed on. For example, [Stack Exchange's app](#) supports Android 4.1+.

ADDITIONAL INFORMATION		
Updated	Installs	Current Version
September 28, 2016	100,000 - 500,000	1.0.89
Requires Android 4.1 and up	Content Rating Rated for 12+ Parental Guidance Recommended Learn more	Interactive Elements Users Interact
Permissions View details	Report Flag as inappropriate	Offered By Stack Exchange

Android Studio will tell you (approximately) what percentage of devices will be supported given the specified minimum SDK.

Lower API levels target more devices but have fewer features available.

When deciding on the **Minimum SDK**, you should consider the [Dashboards stats](#), which will give you version information about the devices that visited the Google Play Store globally in the last week.

Platform Versions

This section provides data about the relative number of devices running a given version of the Android platform.

For information about how to target your application to devices based on platform version, read [Supporting Different Platform Versions](#).



Data collected during a 7-day period ending on February 6, 2017.
Any versions with less than 0.1% distribution are not shown.

From: [Dashboards](#) on Android Developer website.

Add an activity

Now we are going to select a default activity for our application. In Android, an [Activity](#) is a single screen that will be presented to the user. An application can house multiple activities and navigate between them. For this example, choose `Empty Activity` and click next.

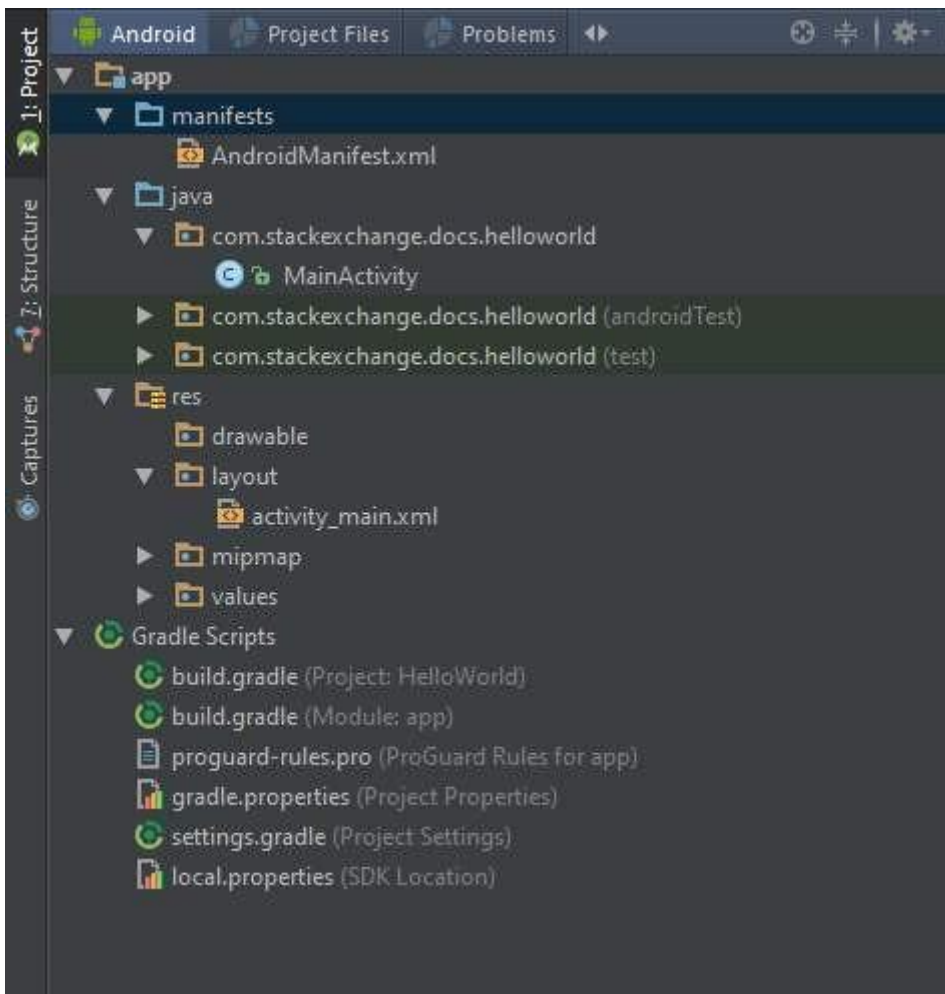
Here, if you wish, you can change the name of the activity and layout. A good practice is to keep `Activity` as a suffix for the activity name, and `activity_` as a prefix for the layout name. If we leave these as the default, Android Studio will generate an activity for us called `MainActivity`, and a layout file called `activity_main`. Now click `Finish`.

Android Studio will create and configure our project, which can take some time depending on the system.

Inspecting the Project

To understand how Android works, let's take a look at some of the files that were created for us.

On the left pane of Android Studio, we can see the [structure of our Android application](#).



First, let's open `AndroidManifest.xml` by double clicking it. The Android manifest file describes some of the basic information about an Android application. It contains the declaration of our activities, as well as some more advanced components.

If an application needs access to a feature protected by a permission, it must declare that it requires that permission with a `<uses-permission>` element in the manifest. Then, when the application is installed on the device, the installer determines whether or not to grant the requested permission by checking the authorities that signed the application's certificates and, in some cases, asking the user. An application can also protect its own components (activities, services, broadcast receivers, and content providers) with permissions. It can employ any of the permissions defined by Android (listed in `android.Manifest.permission`) or declared by other applications. Or it can define its own.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.stackoverflow.helloworld">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

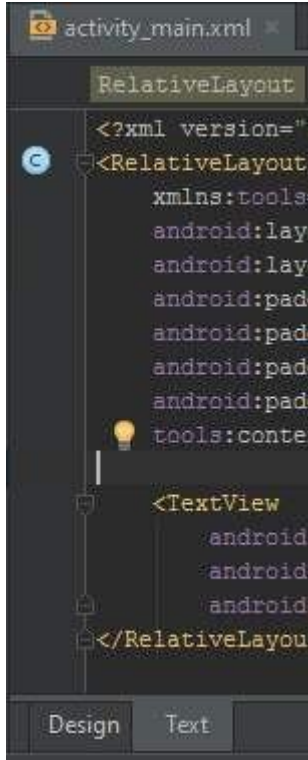
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
```



```
</application>
</manifest>
```

Next, let's open `activity_main.xml` which is located in `app/src/main/res/layout/`. This file contains declarations for the visual components of our MainActivity. You will see visual designer. This allows you to drag and drop elements onto the selected layout.

You can also switch to the xml layout designer by clicking "Text" at the bottom of Android Studio, as seen here:



```
<?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: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.stackexchange.docs.helloworld.MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!" />

</RelativeLayout>
```

You will see a widget called a `TextView` inside of this layout, with the `android:text` property set to "Hello World!". This is a block of text that will be shown to the user when they run the application.

You can read more about [Layouts and attributes](#).

Next, let's take a look at `MainActivity`. This is the Java code that has been generated for `MainActivity`.

```
public class MainActivity extends AppCompatActivity {

    // The onCreate method is called when an Activity starts
```



```

// This is where we will set up our layout
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // setContentView sets the Activity's layout to a specified XML layout
    // In our case we are using the activity_main layout
    setContentView(R.layout.activity_main);
}
}

```

As defined in our Android manifest, `MainActivity` will launch by default when a user starts the `HelloWorld` app.

Lastly, open up the file named `build.gradle` located in `app/`.

Android Studio uses the build system **Gradle** to compile and build Android applications and libraries.

```

apply plugin: 'com.android.application'

android {
    signingConfigs {
        applicationName {
            keyAlias 'applicationName'
            keyPassword 'password'
            storeFile file('../key/applicationName.jks')
            storePassword 'anotherPassword'
        }
    }
    compileSdkVersion 26
    buildToolsVersion "26.0.0"

    defaultConfig {
        applicationId "com.stackexchange.docs.helloworld"
        minSdkVersion 16
        targetSdkVersion 26
        versionCode 1
        versionName "1.0"
        signingConfig signingConfigs.applicationName
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    testCompile 'junit:junit:4.12'
    compile 'com.android.support:appcompat-v7:26.0.0'
}

```

This file contains information about the build and your app version, and you can also use it to add dependencies to external libraries. For now, let's not make any changes.

It is advisable to always select the latest version available for the dependencies:

- [buildToolsVersion](#): 26.0.0
- [com.android.support:appcompat-v7](#): 26.0.0 (July 2017)
- [firebase](#): 11.0.4 (August 2017)

compileSdkVersion

`compileSdkVersion` is your way to tell *Gradle* what version of the Android SDK to compile your app with. Using the new Android SDK is a requirement to use any of the new APIs added in that level.

It should be emphasized that changing your `compileSdkVersion` does not change runtime behavior. While new compiler warnings/errors may be present when changing your `compileSdkVersion`, your `compileSdkVersion` is not included in your APK: it is purely used at compile time.

Therefore it is strongly recommended that you always compile with the latest SDK. You'll get all the benefits of new compilation checks on existing code, avoid newly deprecated APIs, and be ready to use new APIs.

minSdkVersion

If `compileSdkVersion` sets the newest APIs available to you, `minSdkVersion` is the *lower bound* for your app. The `minSdkVersion` is one of the signals the Google Play Store uses to determine which of a user's devices an app can be installed on.

It also plays an important role during development: by default lint runs against your project, warning you when you use any APIs above your `minSdkVersion`, helping you avoid the runtime issue of attempting to call an API that doesn't exist. Checking the system version at runtime is a common technique when using APIs only on newer platform versions.

targetSdkVersion

`targetSdkVersion` is the main way Android provides forward compatibility by not applying behavior changes unless the `targetSdkVersion` is updated. This allows you to use new APIs prior to working through the behavior changes. Updating to target the latest SDK should be a high priority for every app. That doesn't mean you have to use every new feature introduced nor should you blindly update your `targetSdkVersion` without testing.

`targetSdkVersion` is the version of Android which is the upper-limit for the available tools. If `targetSdkVersion` is less than 23, the app does not need to request permissions at runtime for an instance, even if the app is being run on API 23+. `targetSdkVersion` does **not** prevent android versions above the picked Android version from running the app.

You can find more info about the Gradle plugin:

- A basic example
- Introduction to the Gradle plugin for android and the wrapper
- Introduction to the configuration of the build.gradle and the DSL methods

Running the Application

Now, let's run our HelloWorld application. You can either run an Android Virtual Device (which you can set up by using the AVD Manager in Android Studio, as described in the example below) or connect your own Android device through a USB cable.

Setting up an Android device

To run an application from Android Studio on your Android Device, you must enable `USB Debugging` in the `Developer Options` in the settings of your device.

Settings > Developer options > USB debugging

If `Developer Options` is not visible in the settings, navigate to `About Phone` and tap on the `Build Number` seven

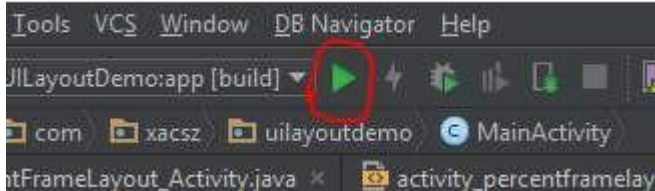
times. This will enable `Developer Options` to show up in your settings.

`Settings > About phone > Build number`

You also might need to change `build.gradle` configuration to build on a version that your device has.

Running from Android Studio

Click the green `Run` button from the toolbar at the top of Android Studio. In the window that appears, select whichever device you would like to run the app on (start an Android Virtual Device if necessary, or see [Setting up an AVD \(Android Virtual Device\)](#) if you need to set one up) and click `OK`.



On devices running Android 4.4 (KitKat) and possibly higher, a pop-up will be shown to authorize USB debugging. Click `OK` to accept.

The application will now install and run on your Android device or emulator.

APK file location

When you prepare your application for release, you configure, build, and test a release version of your application. The configuration tasks are straightforward, involving basic code cleanup and code modification tasks that help optimize your application. The build process is similar to the debug build process and can be done using JDK and Android SDK tools. The testing tasks serve as a final check, ensuring that your application performs as expected under real-world conditions. When you are finished preparing your application for release you have a signed APK file, which you can distribute directly to users or distribute through an application marketplace such as Google Play.

Android Studio

Since in the above examples Gradle is used, the location of the generated APK file is: `<Your Project Location>/app/build/outputs/apk/app-debug.apk`

IntelliJ

If you are a user of IntelliJ before switching to Studio, and are importing your IntelliJ project directly, then nothing changed. The location of the output will be the same under:

```
out/production/...
```

Note: this is will become deprecated sometimes around 1.0

Eclipse

If you are importing Android Eclipse project directly, do not do this! As soon as you have dependencies in your project (jars or Library Projects), this will not work and your project will not be properly setup. If you have no dependencies, then the apk would be under the same location as you'd find it in Eclipse:

```
bin/...
```

Section 1.2: Setting up Android Studio

[Android Studio](#) is the Android development IDE that is officially supported and recommended by Google. Android Studio comes bundled with the [Android SDK Manager](#), which is a tool to download the `Android` SDK components required to start developing apps.

Installing Android Studio and `Android` SDK tools:

1. Download and install [Android Studio](#).
2. Download the latest SDK Tools and SDK Platform-tools by opening the Android Studio, and then following the [Android SDK Tool Updates](#) instructions. You should install the latest available stable packages.

If you need to work on old projects that were built using older SDK versions, you may need to download these versions as well

Since Android Studio 2.2, a copy of the latest OpenJDK comes bundled with the install and is the [recommended JDK](#) (Java Development Kit) for all Android Studio projects. This removes the requirement of having Oracle's JDK package installed. To use the bundled SDK, proceed as follows;

1. Open your project in Android Studio and select **File > Project Structure** in the menu bar.
2. In the **SDK Location** page and under **JDK location**, check the **Use embedded JDK** checkbox.
3. Click **OK**.

Configure Android Studio


Android Studio provides access to two configuration files through the **Help** menu:

- [studio.vmoptions](#): Customize options for Studio's Java Virtual Machine (JVM), such as heap size and cache size. Note that on Linux machines this file may be named `studio64.vmoptions`, depending on your version of Android Studio.
- [idea.properties](#): Customize Android Studio properties, such as the plugins folder path or maximum supported file size.

Change/add theme

You can change it as your preference. `File->Settings->Editor->Colors & Fonts->` and select a theme. Also you can download new themes from <http://color-themes.com/>. Once you have downloaded the `.jar.zip` file, go to `File -> Import Settings...` and choose the file downloaded.

Compiling Apps

Create a new project or open an existing project in Android Studio and press the green Play button  on the top toolbar to run it. If it is gray you need to wait a second to allow Android Studio to properly index some files, the progress of which can be seen in the bottom status bar.

If you want to create a project from the shell make sure that you have a `local.properties` file, which is created by Android Studio automatically. If you need to create the project without Android Studio you need a line starting with `sdk.dir=` followed by the path to your SDK installation.

Open a shell and go into the project's directory. Enter `./gradlew aR` and press enter. `aR` is a shortcut for `assembleRelease`, which will download all dependencies for you and build the app. The final APK file will be in `ProjectName/ModuleName/build/outputs/apk` and will be called `ModuleName-release.apk`.

Section 1.3: Android programming without an IDE

This is a minimalist Hello World example that uses only the most basic Android tools.

Requirements and assumptions

- Oracle JDK 1.7 or later
- Android SDK Tools (just the [command line tools](#))

This example assumes Linux. You may have to adjust the syntax for your own platform.

Setting up the Android SDK

After unpacking the SDK release:

1. Install additional packages using the SDK manager. Don't use `android update sdk --no-ui` as instructed in the bundled `Readme.txt`; it downloads some 30 GB of unnecessary files. Instead use the interactive SDK manager `android sdk` to get the recommended minimum of packages.
2. Append the following JDK and SDK directories to your execution PATH. This is optional, but the instructions below assume it.
 - JDK/bin
 - SDK/platform-tools
 - SDK/tools
 - SDK/build-tools/LATEST (*as installed in step 1*)
3. Create an Android virtual device. Use the interactive AVD Manager (`android avd`). You might have to fiddle a bit and search for advice; the [on-site instructions](#) aren't always helpful.

(You can also use your own device)

4. Run the device:

```
emulator -avd DEVICE
```

5. If the device screen appears to be locked, then swipe to unlock it.

Leave it running while you code the app.

Coding the app

0. Change to an empty working directory.
1. Make the source file:

```
mkdir --parents src/dom/domain  
touch src/dom/domain/SayingHello.java
```

Content:

```
package dom.domain;  
import android.widget.TextView;
```

```
public final class SayingHello extends android.app.Activity
{
    protected @Override void onCreate( final android.os.Bundle activityState )
    {
        super.onCreate( activityState );
        final TextView textV = new TextView( SayingHello.this );
        textV.setText( "Hello world" );
        setContentView( textV );
    }
}
```

2. Add a manifest:

```
touch AndroidManifest.xml
```

Content:

```
<?xml version='1.0'?>
<manifest xmlns:a='http://schemas.android.com/apk/res/android'
package='dom.domain' a:versionCode='0' a:versionName='0'>
    <application a:label='Saying hello'>
        <activity a:name='dom.domain.SayingHello'>
            <intent-filter>
                <category a:name='android.intent.category.LAUNCHER' />
                <action a:name='android.intent.action.MAIN' />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

3. Make a sub-directory for the declared resources:

```
mkdir res
```

Leave it empty for now.

Building the code

0. Generate the source for the resource declarations. Substitute here the correct path to your **SDK**, and the installed **API** to build against (e.g. "android-23"):

```
aapt package -f \
-I SDK/platforms/android-API/android.jar \
-J src -m \
-M AndroidManifest.xml -S res -v
```

Resource declarations (described further below) are actually optional. Meantime the above call does nothing if res/ is still empty.

1. Compile the source code to Java bytecode (.java → .class):

```
javac \
-bootclasspath SDK/platforms/android-API/android.jar \
-classpath src -source 1.7 -target 1.7 \
src/dom/domain/*.java
```

2. Translate the bytecode from Java to Android (.class → .dex):

First using Jill (.class → .jayce):

```
java -jar SDK/build-tools/LATEST/jill.jar \\  
  --output classes.jayce src
```

Then Jack (.jayce → .dex):

```
java -jar SDK/build-tools/LATEST/jack.jar \\  
  --import classes.jayce --output-dex .
```

Android bytecode used to be called "Dalvik executable code", and so "dex".

You could replace steps 11 and 12 with a single call to Jack if you like; it can compile directly from Java source (.java → .dex). But there are advantages to compiling with `javac`. It's a better known, better documented and more widely applicable tool.

3. Package up the resource files, including the manifest:

```
aapt package -f \\  
  -F app.apkPart \\  
  -I SDK/platforms/android-API/android.jar \\  
  -M AndroidManifest.xml -S res -v
```

That results in a partial APK file (Android application package).

4. Make the full APK using the `ApkBuilder` tool:

```
java -classpath SDK/tools/lib/sdklib.jar \\  
  com.android.sdklib.build.ApkBuilderMain \\  
  app.apkUnalign \\  
  -d -f classes.dex -v -z app.apkPart
```

It warns, "THIS TOOL IS DEPRECATED. See --help for more information." If `--help` fails with an `ArrayIndexOutOfBoundsException`, then instead pass no arguments:

```
java -classpath SDK/tools/lib/sdklib.jar \\  
  com.android.sdklib.build.ApkBuilderMain
```

It explains that the CLI (`ApkBuilderMain`) is deprecated in favour of directly calling the Java API (`ApkBuilder`). (If you know how to do that from the command line, please update this example.)

5. Optimize the data alignment of the APK ([recommended practice](#)):

```
zipalign -f -v 4 app.apkUnalign app.apk
```

Installing and running

0. Install the app to the Android device:


```
adb install -r app.apk
```

1. Start the app:

```
adb shell am start -n dom.domain/.SayingHello
```

It should run and say hello.

That's all. That's what it takes to say hello using the basic Android tools.

Declaring a resource

This section is optional. Resource declarations aren't required for a simple "hello world" app. If they aren't required for your app either, then you could streamline the build somewhat by omitting step 10, and removing the reference to the res/ directory from step 13.

Otherwise, here's a brief example of how to declare a resource, and how to reference it.

0. Add a resource file:

```
mkdir res/values  
touch res/values/values.xml
```

Content:

```
<?xml version='1.0'?>  
<resources>  
  <string name='appLabel'>Saying hello</string>  
</resources>
```

1. Reference the resource from the XML manifest. This is a declarative style of reference:

```
<!-- <application a:label='Saying hello'> -->  
  <application a:label='@string/appLabel'>
```

2. Reference the same resource from the Java source. This is an imperative reference:

```
// v.setText( "Hello world" );  
v.setText( "This app is called "  
  + getResources().getString( R.string.appLabel ) );
```

3. Test the above modifications by rebuilding, reinstalling and re-running the app (steps 10-17).

It should restart and say, "This app is called Saying hello".

Uninstalling the app

```
adb uninstall dom.domain
```

See also

- [original question](#) - The original question that prompted this example
- [working example](#) - A working build script that uses the above commands

Section 1.4: Application Fundamentals

Android Apps are written in Java. The Android SDK tools compile the code, data and resource files into an APK (Android package). Generally, one APK file contains all the content of the app.

Each app runs on its own virtual machine(VM) so that app can run isolated from other apps. Android system works with the principle of least privilege. Each app only has access to the components which it requires to do its work, and no more. However, there are ways for an app to share data with other apps, such as by sharing Linux user id between app, or apps can request permission to access device data like SD card, contacts etc.

App Components

App components are the building blocks of an Android app. Each components plays a specific role in an Android app which serves a distinct purpose and has distinct life-cycles(the flow of how and when the component is created and destroyed). Here are the four types of app components:

1. **Activities:** An activity represents a single screen with a User Interface(UI). An Android app may have more than one activity. (e.g. an email app might have one activity to list all the emails, another to show the contents of each email, and another to compose new email.) All the activities in an App work together to create a User eXperience (UX).
2. **Services:** A service runs in the background to perform long-running operations or to perform work for a remote processes. A service does not provide any UI, it runs only in the background with the User's input. (e.g. a service can play music in the background while the user is in a different App, or it might download data from the internet without blocking user's interaction with the Android device.)
3. **Content Providers:** A content provider manages shared app data. There are four ways to store data in an app: it can be written to a file and stored in the file system, inserted or updated to a SQLite database, posted to the web, or saved in any other persistent storage location the App can access. Through content providers, other Apps can query or even modify the data. (e.g. Android system provides a content provider that manages the user's contact information so that any app which has permission can query the contacts.) Content providers can also be used to save the data which is private to the app for better data integrity.
4. **Broadcast receivers:** A broadcast receiver responds to the system-wide broadcasts of announcements (e.g. a broadcast announcing that the screen has turned off, the battery is low, etc.) or from Apps (e.g. to let other apps know that some data has been downloaded to the device and is available for them to use). Broadcast receivers don't have UIs but they can show notification in the status bar to alert the user. Usually broadcast receivers are used as a gateway to other components of the app, consisting mostly of activities and services.

One unique aspect of the Android system is that any app can start another app's component (e.g. if you want to make call, send SMS, open a web page, or view a photo, there is an app which already does that and your app can make use of it, instead of developing a new activity for the same task).

When the system starts a component, it starts the process for that app (if it isn't already running, i.e. only one foreground process per app can run at any given time on an Android system) and instantiates the classes needed for that component. Thus the component runs on the process of that App that it belongs to. Therefore, unlike apps on other systems, Android apps don't have a single entry point(there is no `main()` method).

Because the system runs each app in a separate process, one app cannot directly activate another app's components, however the Android system can. Thus to start another app's component, one app must send a message to the system that specifies an intent to start that component, then the system will start that component.

Context

Instances of the class `android.content.Context` provide the connection to the Android system which executes the application. Instance of Context is required to get access to the resources of the project and the global information

about the app's environment.

Let's have an easy to digest example: Consider you are in a hotel, and you want to eat something. You call room-service and ask them to bring you things or clean up things for you. Now think of this hotel as an Android app, yourself as an activity, and the room-service person is then your context, which provides you access to the hotel resources like room-service, food items etc.

Yet another example, You are in a restaurant sitting on a table, each table has an attendant, whenever you want to order food items you ask the attendant to do so. The attendant then places your order and your food items get served on your table. Again in this example, the restaurant is an Android App, the tables or the customers are App components, the food items are your App resources and the attendant is your context thus giving you a way to access the resources like food items.

Activating any of the above components requires the context's instance. Not just only the above, but almost every system resource: creation of the UI using views (discussed later), creating instance of system services, starting new activities or services -- all require context.

More detailed description is written [here](#).

Section 1.5: Setting up an AVD (Android Virtual Device)

TL;DR It basically allows us to simulate real devices and test our apps without a real device.

According to [Android Developer Documentation](#),

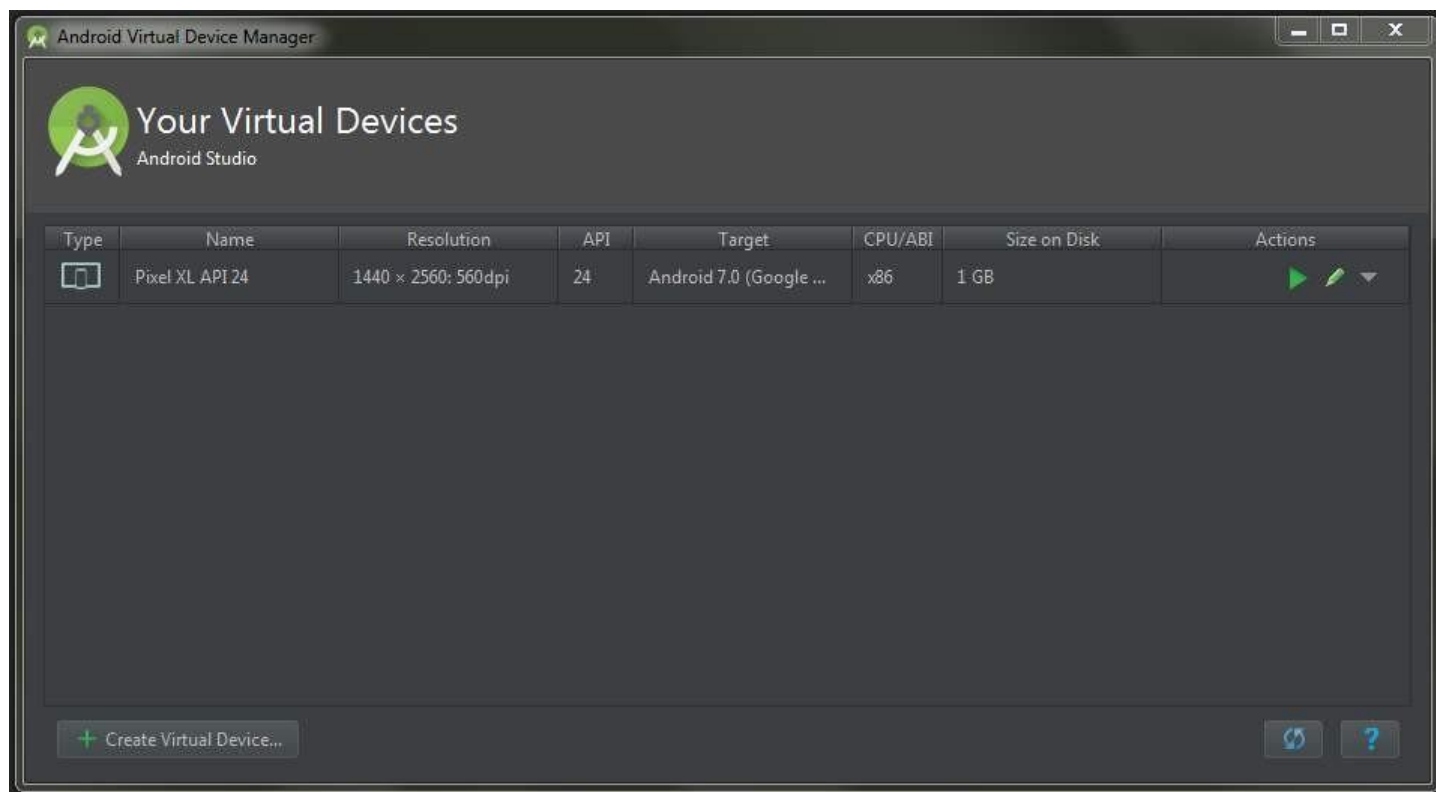
an **Android Virtual Device (AVD)** *definition* lets you define the characteristics of an Android Phone, Tablet, Android Wear, or Android TV device that you want to simulate in the Android Emulator. The AVD Manager helps you easily create and manage AVDs.

To set up an AVD, follow these steps:

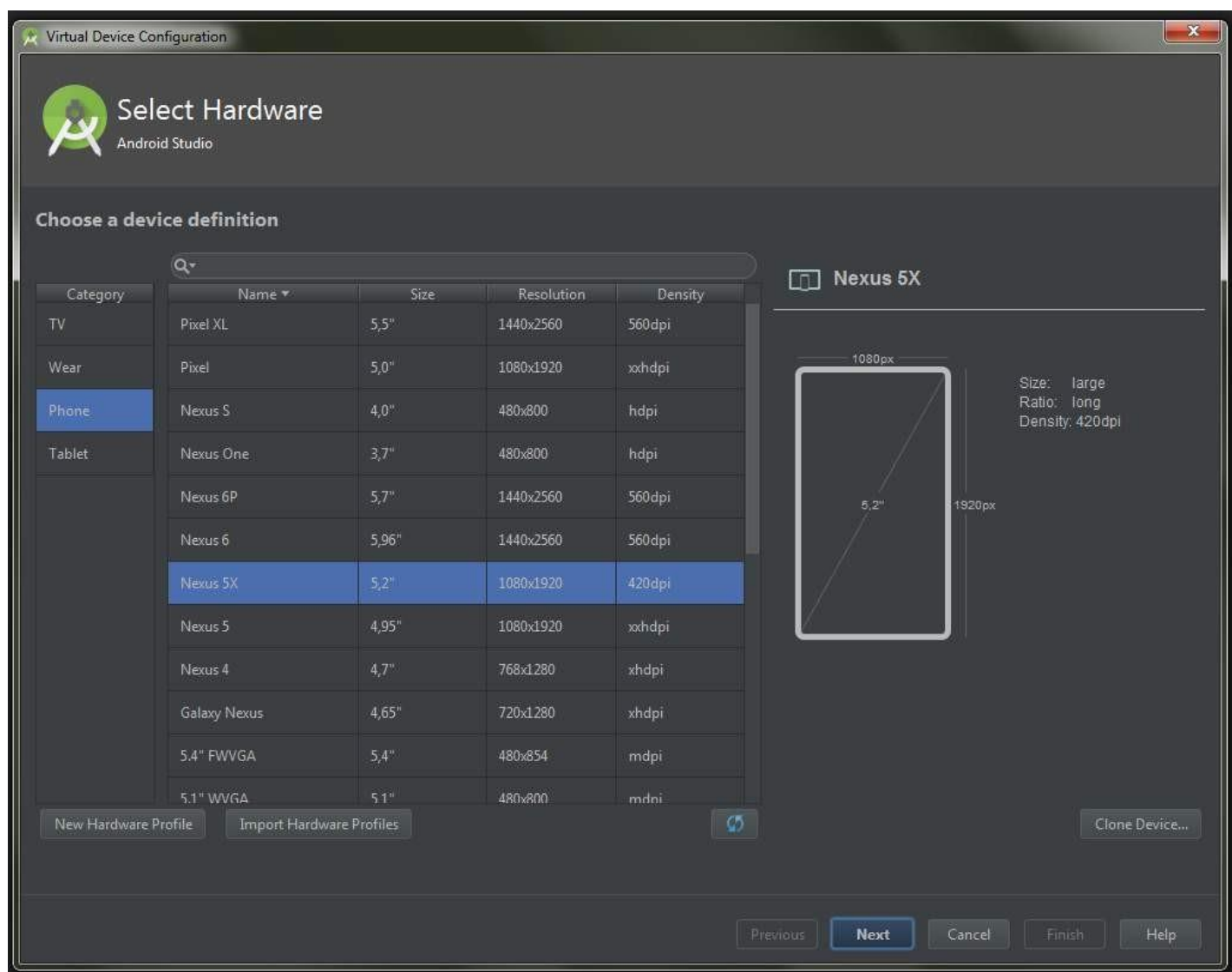
1. Click this button to bring up the AVD Manager:



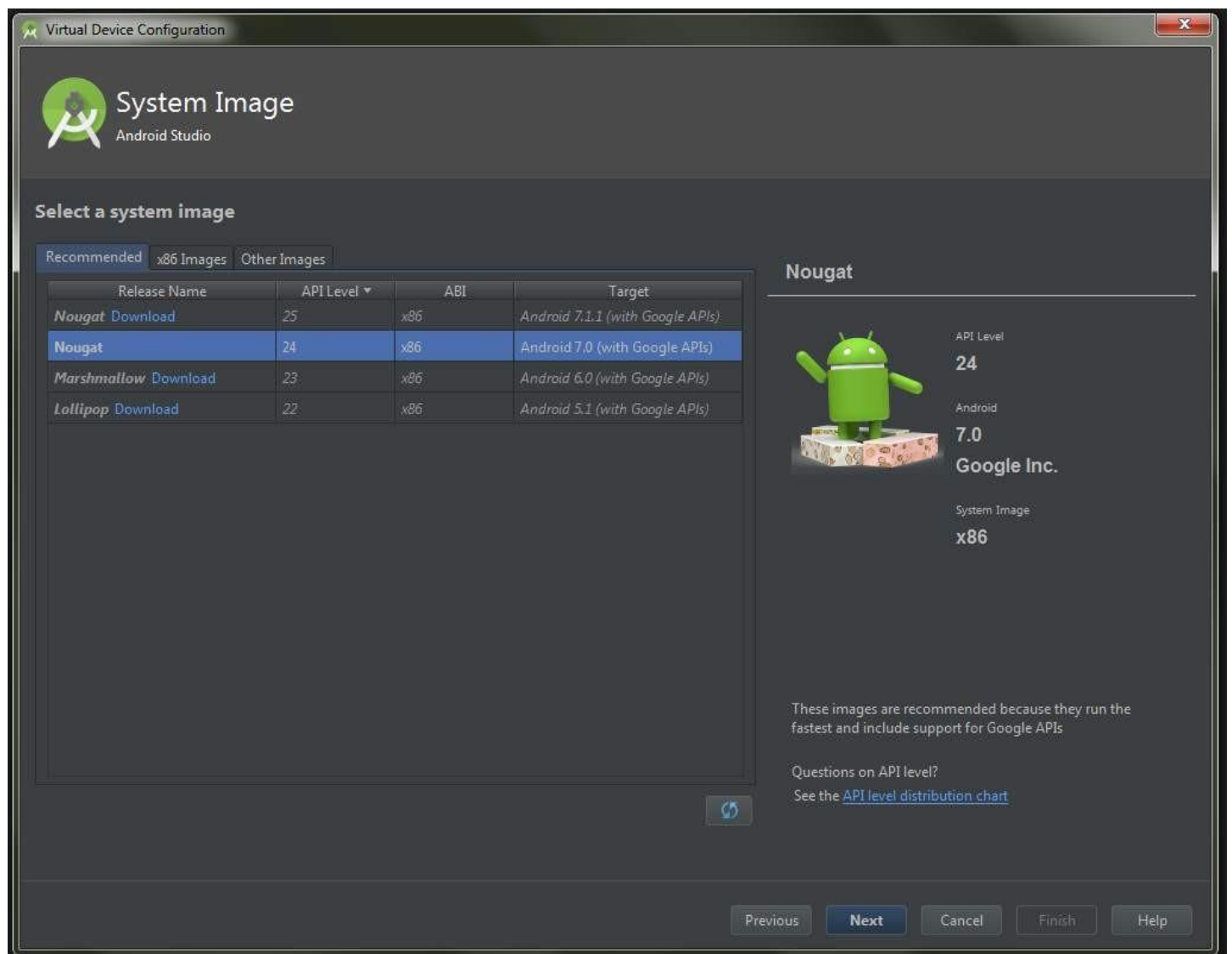
2. You should see a dialog like this:



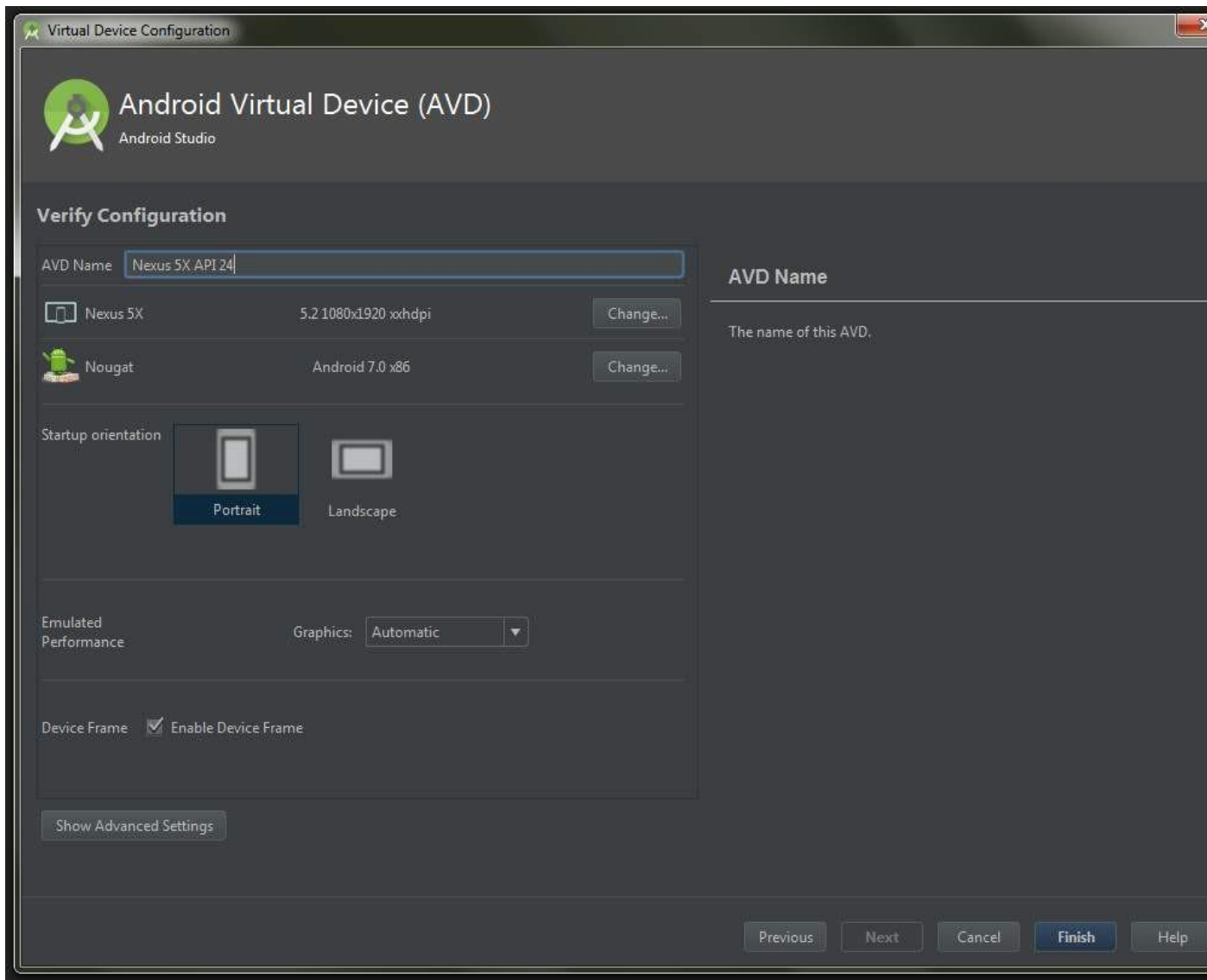
3. Now click the + Create Virtual Device... button. This will bring up Virtual Device Configuration Dialog:



4. Select any device you want, then click **Next**:



5. Here you need to choose an Android version for your emulator. You might also need to download it first by clicking **Download**. After you've chosen a version, click **Next**.



6. Here, enter a name for your emulator, initial orientation, and whether you want to display a frame around it. After you chosen all these, click **Finish**.

7. Now you got a new AVD ready for launching your apps on it.

Type	Name	Resolution	API	Target	CPU/ABI	Size on Disk	Actions
	Nexus 5X API 24	1080 × 1920: 420dpi	24	Android 7.0 (Google ...	x86	650 MB	