

Barbara Hohensee



Getting
Started
with

ANDROID
studio

Impressum

Text: © Copyright by Barbara Hohensee

Barbara Hohensee

Utlandagatan 33

41261 Gothenburg

Sweden

greendogdevelop@gmail.com

All rights reserved.

Publication date August 2013

Last Update February, 2014

Blog: <http://google-android-studio.blogspot.com>

About Barbara Hohensee



Barbara Hohensee has worked more than 10 years as a network administrator. Among other things for DaimlerChrysler research.

She is familiar to the most used operating systems such as Unix, Linux, Windows, Mac OS, Android and iOS.

The author has further, in addition to courses for network and Linux developed and teaches courses for Perl and Java programming.

Currently, she devotes herself to her new great love of android programming

Table of Contents

[Preamble](#)
[About Android Studio](#)
[Installation and Configuration](#)
[SDK Manager](#)
[AVD's](#)
[Starting a new Android Project](#)
[Overview IDE](#)
[Project Structur](#)
[Gradle Build System](#)
[Projekt Configuration](#)
[Creating Layouts](#)
[Activities](#)
[Build & Run](#)
[Debugging](#)
[Testing](#)
[Preparing the App for the Android Market](#)
[Importing Projects](#)
[Google Cloud Endpoints](#)
[Google Play Service/ Maps](#)
[Product Flavors - Build Types - Build Variants](#)
[Game development with AndEngine](#)
[Developing for Google TV](#)
[Android Code Templates](#)
[Links](#)
[Impressum](#)
[About the Author](#)

Preamble

This book will help you to familiar yourself with the new IDE for Android development called Android Studio.

Who should read this book? Because Android Studio is based on IntelliJ, the book will be interesting for everybody who hasn't worked with IntelliJ yet.

There are a lots of screenshots to make it as easy as possible. So even a beginner in Android development gets a chance to understand how Android Studio works.

This book will help you to accomplish the most common tasks.

What this book covers:

- Installation of Android Studio
- Creating a new Android App Project with Google's ActionBar
- Android SDK Manager
- AVD's
- Overview IDE and Editor
- The new project structure of an Android app project
- Overview Gradle Build System
- Local installation of Gradle
- Local installation of Maven
- Building an .aar library
- Creating a local Maven repository
- Using the .aar library
- Version Control (VCS)
- Working with Activities and Layouts
- Build and run the app

- Debugging
- Testing: Creating and running Test projects
- Preparing the app for the Android Market
- Import of an Android Project
-
- o Android Studio
- o Eclipse
- o GitHub
- Google Cloud Endpoints
- Google Play Service SDK, Google Maps v2
- Product Flavors - Build Types - Build Variants
- Game development
-
- o libGDX setup to develop, run and deploy Desktop an Android games
- o AndEngine setup and example project

About Android Studio

Google unveiled at the developer conference, the new development environment for android app development. The new IDE based on IntelliJ will soon replace Eclipse. At the same time there will be change for the build system too. Ant as the build system has been replaced in Android Studio to Gradle.

One of the core pieces of Android Studio is the powerful code editor with built in features like "Smart Editing", to provide for more readable code or the "Advanced code refactoring".

Another highlight of Android Studio is of course the new build system based on Gradle.

Gradle allows the developer to apply different configurations of the same code, to produce different versions of the same application code. This is useful, among other things, if you want to give out a free and a paid version of an app.

Generally Gradle improves the reusability of code and integration on a build server.

Like Eclipse has Android Studio a graphical and a text user interface for designing the app layout_

Both the design mode and the text mode of the editor have improved. The editor is now showing a live preview of the layout for different resolutions, Android versions and country-specific characteristics.

Android studio got some new services and integrated which makes it easier to handle translation or to connect with Google Cloud Messaging (CGM) that allows to sending messages to the app and receiving messages from apps on cloud servers.

Google develops Android Studio in collaboration with JetBrains, based on the community version of IntelliJ. JetBrains IntelliJ Java IDE has support for Android app development for 2 years. In the current version of IntelliJ 12 the innovations that have emerged from the collaboration with Google are not yet integrated. They will be integrated in version 13, release date December 2013. The new version will continue to have the support for Java, Android, Adobe Gaming SDK Groovy, Scala.

The Android studio, however, will be limited to Android app development. Google has still no official release date for the Android studio out.

The current version of Android studios can be downloaded from here:

<http://developer.android.com/sdk/installing/studio.html>

Is Android Studio ready for production? Are you ready for Android Studio?

Many people like to start with Android Studio and ask themselves if Android Studio is ready for production.

There is no Yes or No answer for this question because it depends on what kind of App you're doing. Maybe not every feature YOU need is already implemented but a lot of the functionality you need to have for Android App Development is already in place.

You can see the current development status at

<http://tools.android.com/download/studio/canary/latest>

All the apps for the book are of course made with Android Studio and my own production has already moved to Android Studio.

Installation of Android Studio

The Android Studio includes the Android SDK with the latest Android platform. The prerequisite for Android Studio is Oracle Java SDK 1.6 or 1.7. In most cases Java SDK 1.6 works better.

After downloading, unpacking and putting it on a place you like, you are not done. Before you can start with an Android project, Android Studio has to be configured.

Installation of Android Studio

The newest version of Android Studio can be downloaded from here:
<http://developer.android.com/sdk/installing/studio.html>

Windows:

Launch the downloaded EXE file, `android-studio-bundle-<version>.exe`.

Follow the setup wizard to install Android Studio.

Known issue: On some Windows systems, the launcher script does not find where Java is installed. If you encounter this problem, you need to set an environment variable indicating the correct location.

Select Start menu -> Computer -> System Properties -> Advanced System Properties. Then open Advanced tab -> Environment Variables and add a new system variable `JAVA_HOME` that points to your JDK folder, for example `C:\Program Files\Java\jdk1.7.0_21`.

Mac OS X:

Open the downloaded DMG file, `android-studio-bundle-<version>.dmg`.

Drag and drop Android Studio into the Applications folder.

Known issue: Depending on your security settings, when you attempt to open Android Studio, you might see a warning that says the package is damaged and should be moved to the trash. If this happens, go to System Preferences > Security & Privacy and under Allow applications downloaded from, select Anywhere. Then open Android Studio again.

Linux:

Unpack the downloaded Tar file, `android-studio-bundle-<version>.tgz`, into an appropriate location for your applications.

To launch Android Studio, navigate to the android-studio/bin/ directory in a terminal and execute studio.sh.

You may want to add android-studio/bin/ to your PATH environmental variable so that you can start Android Studio from any directory.

Android Studio Configuration

The configuration is on all operation systems the same. Only the PATH will be different.

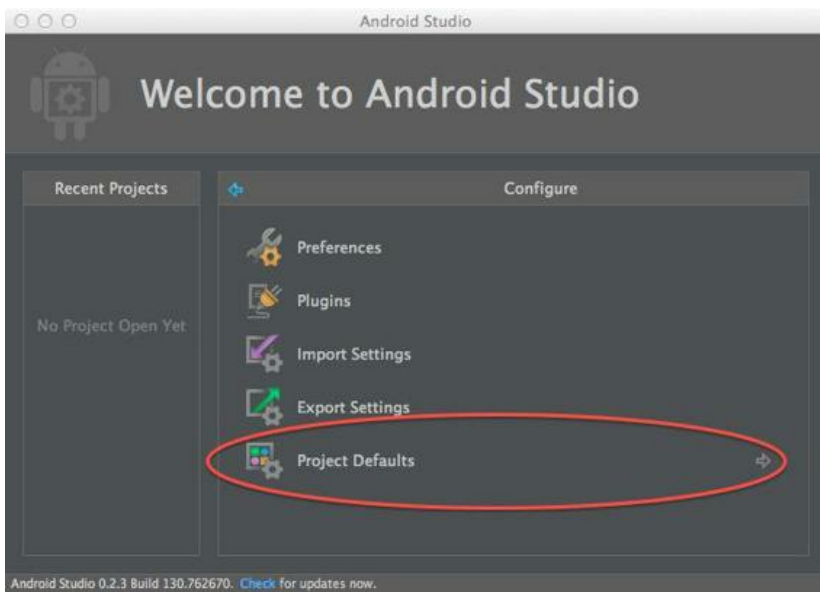
The downloaded package of Android Studio includes the latest Android SDK. Every other Android SDK has to be downloaded using the SDK Manager (Chapter 3)

Before you can start with your first Android project, Android Studio has to be configured. If you need to change the configuration later, you have to go to the menu "Project Structure" again.

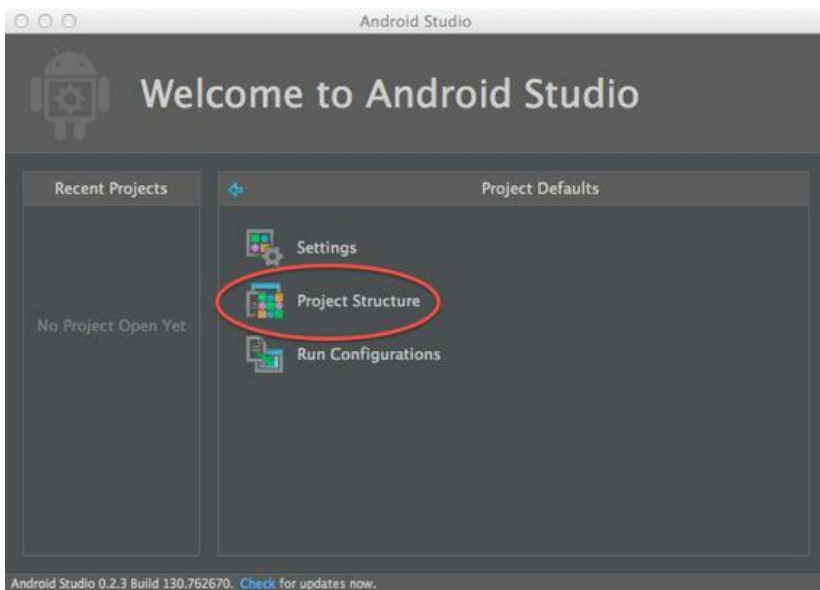
Start Android Studio and click on "Configure" in the Welcome screen.



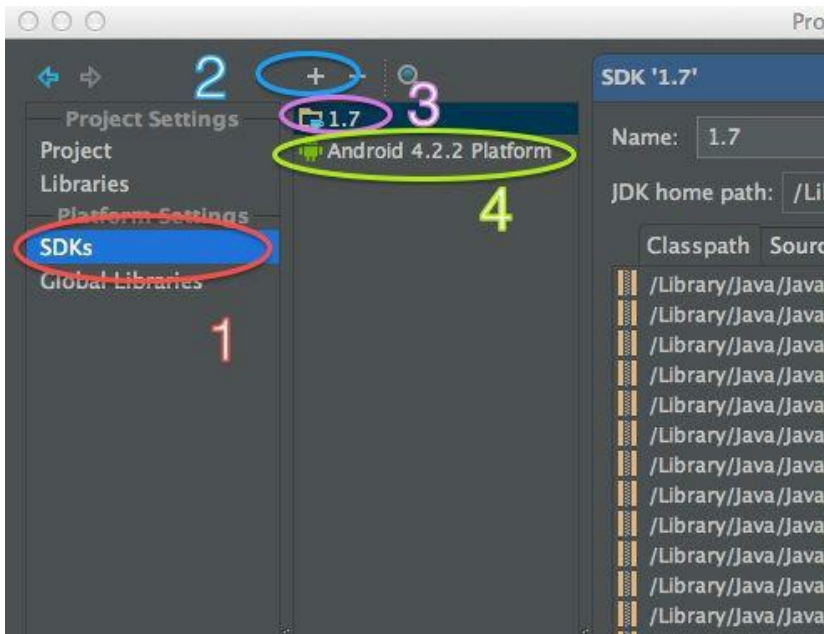
Inside the "Configure Menus" click on "Project Defaults"



From the "Project Defaults" screen choose "Project Structure"



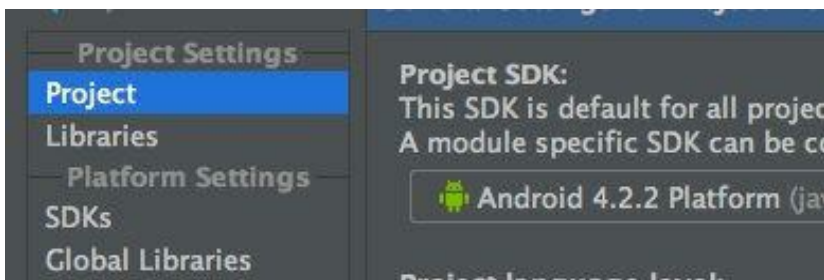
In "Project Structure", choose "SDKs" from the left Panel, go to the Panel in the middle and click the green <+> and take the "JDK" first, before the Android Platform.



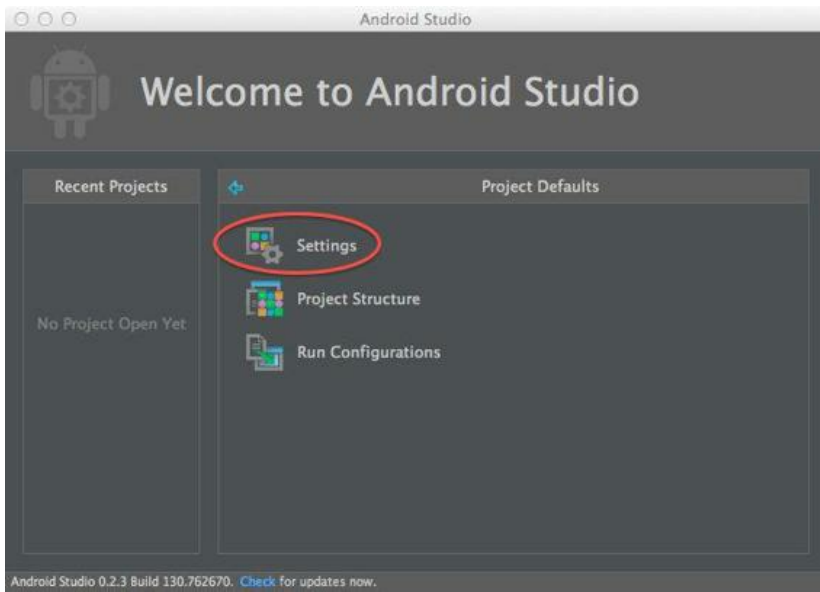
The right panel should show you the folder where the Java SDK installed is. If not, you properly didn't set the `JAVA_HOME` correct. This procedure connected the Java SDK to Android Studio.

In the same way you connect the Android SDK with Android Studio. When it comes to choose the folder, navigate to the folder where Android Studio is installed and go to the underlying sdk folder.

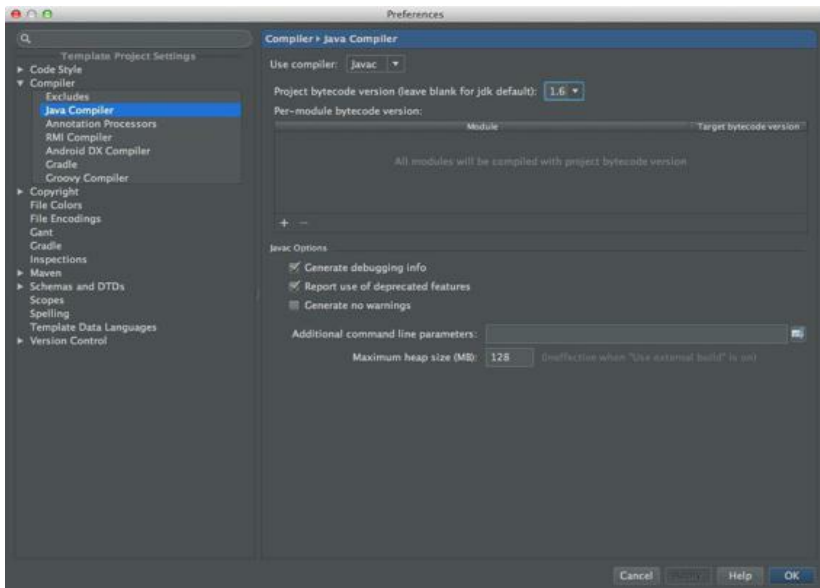
In the next step move to the left panel called "Project Settings". Under the menu "Project", you have to decide which Android Version should be used as the default SDK.



Back to the screen „Project Defaults“ and „Settings“

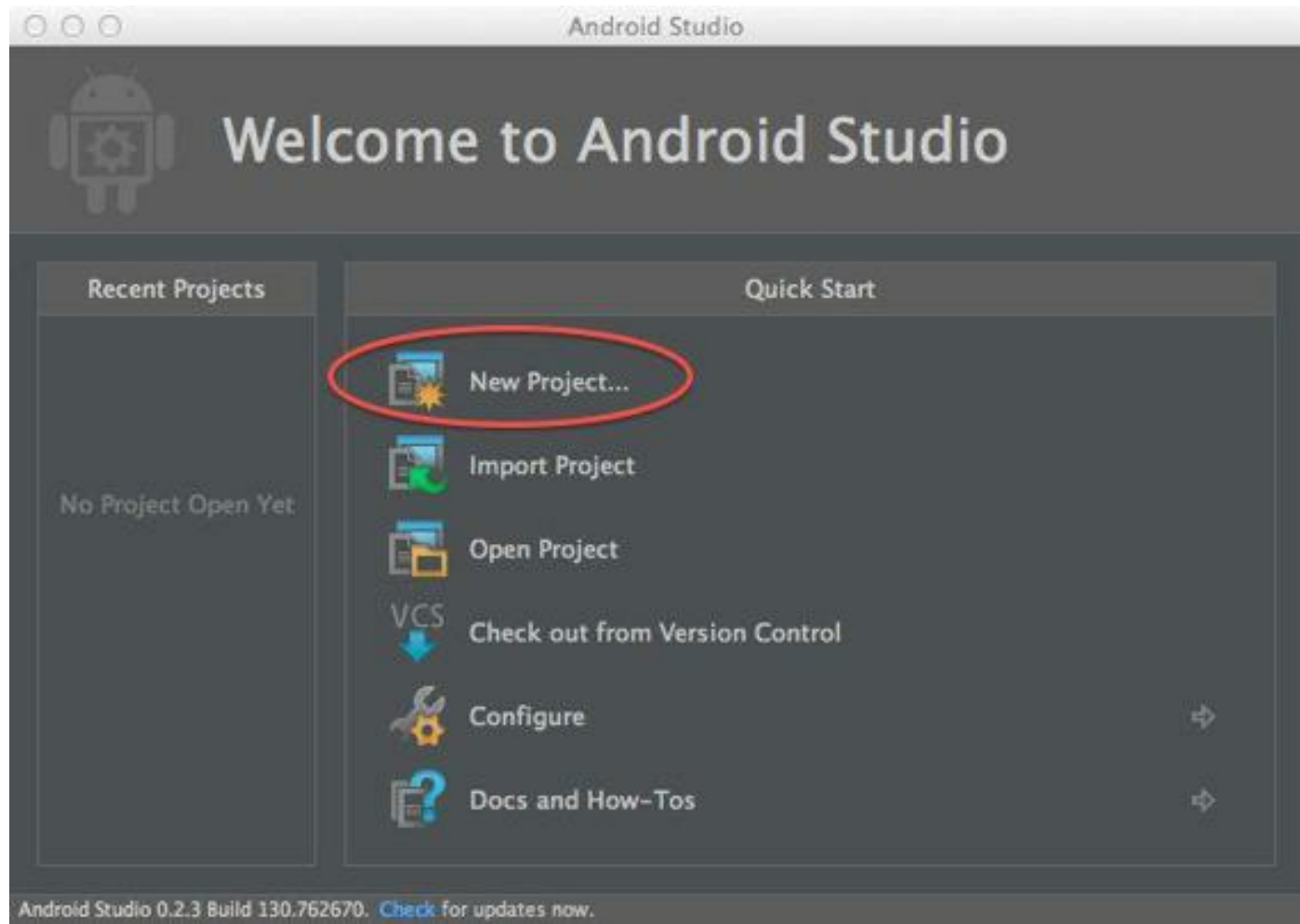


Check out that the right Java version is chosen.



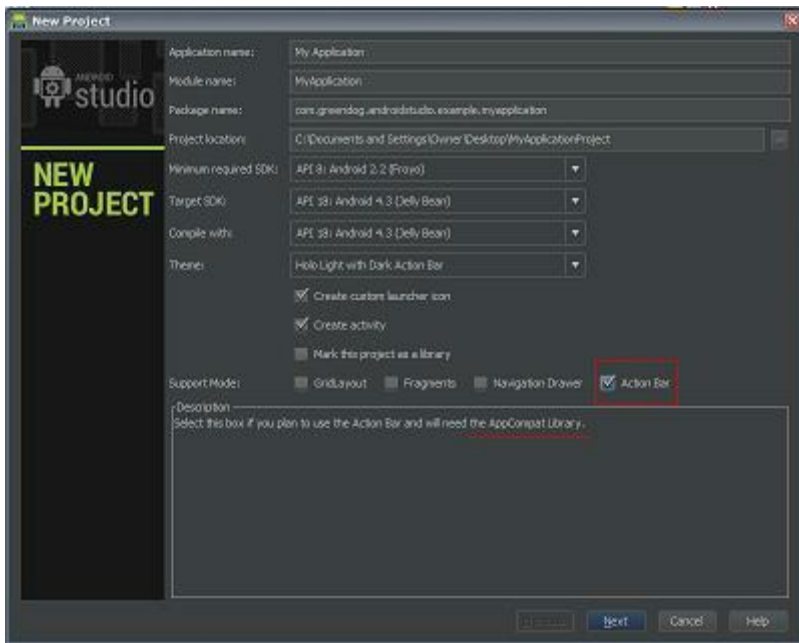
Creating a new Android App Project

When Android Studio after the installation opens, it will show the Welcome screen.



Start creating a new Android Project by clicking on "New Project" from the top of the list.

This will open the New Project Wizard



Application name:

The Application name is name you will see in the title of your application and for example in Google Play.

The name will be stored in "strings.xml" in the variable "app_name". No spaces in the name.

Module name:

Role, no spaces.

Package name:

The id for the App. This name will be added in all Java files and the AndroidManifest.xml.

Project location:

The Project location can be anywhere, any directory where you have the rights to write.

Minimum required SDK:

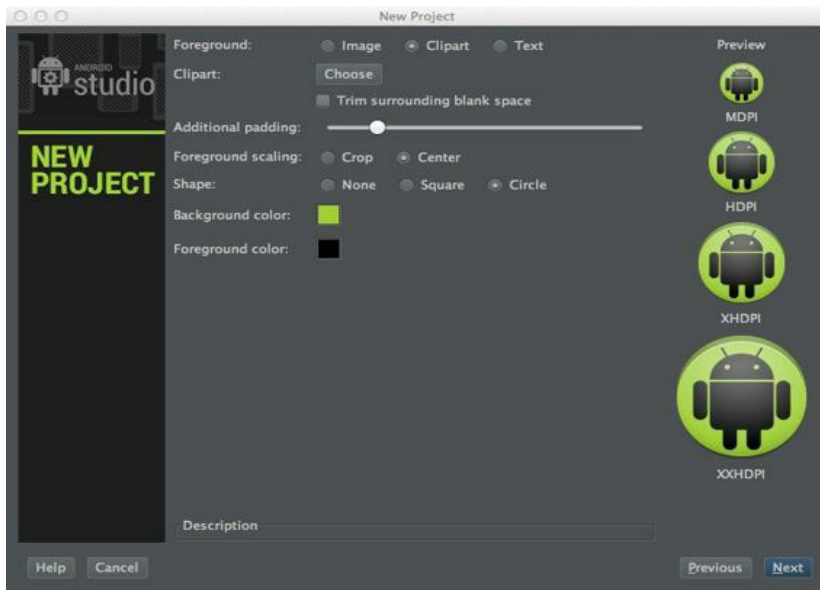
As Standard Android Studio gives you API 7, Android 2.1. In many cases you need minimum API 8 for example when using Ads Provider, like Google's Admob.

Support Mode:

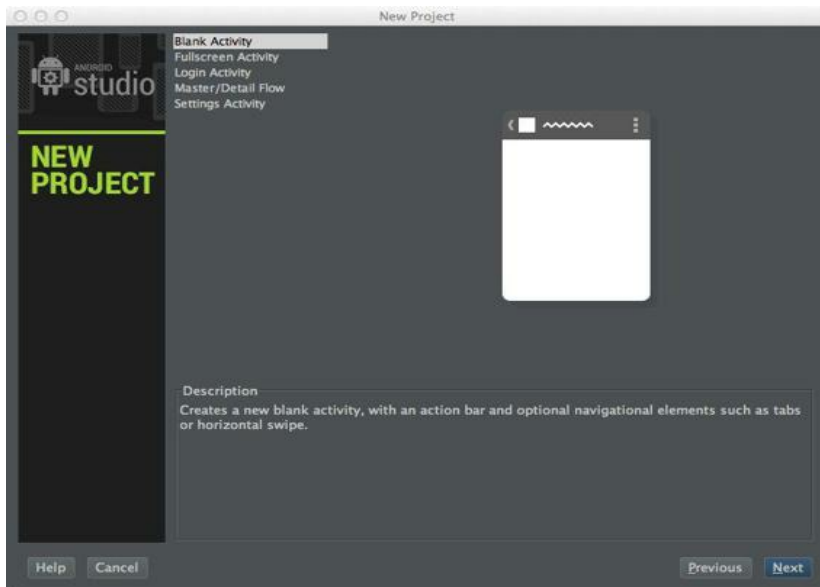
By choosing one of the support modes, Android Studio adds the appropriate support library to the project.

In this case we activated **ActionBar** to import the support library v7 appcompat.

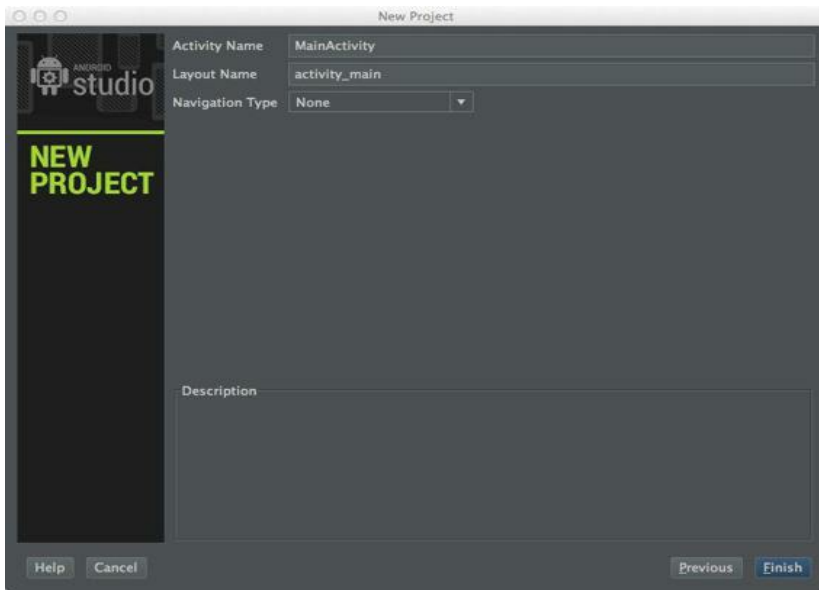
The next screen let you choose and customize the App icon.



With <Next> you can select the Activity template of your choice. In this example, the "Blank Activity" is chosen, which produces the "Hello World" App.



In the next screen write the „Activity Name“ and „Layout Name“ for the first Activity and it's layout file.



From Android Studio 3.0 this screen has changed. By default Android Studio will now create a Fragment Layout instead of the single-screen layout.

You can force Android Studio to create the old style single-layout by changing the name from the Frame Layout to the same as the Main Layout. And you may want to change the MainActivity.java into this: (after the project has build up themself)

```
package com.greendog.androidstudio.example.myapplication;

import android.support.v7.app.ActionBarActivity;
import android.support.v7.app.ActionBar;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;

public class MainActivity extends ActionBarActivity {

    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        // Get the view from activity_main.xml

        setContentView(R.layout.activity_main);

    }

    @Override

    public boolean onCreateOptionsMenu(Menu menu) {
```

```
        // Inflate the menu; this adds items to the action bar if it is
present.

        getMenuInflater().inflate(R.menu.main, menu);

        return true;
    }
}
```

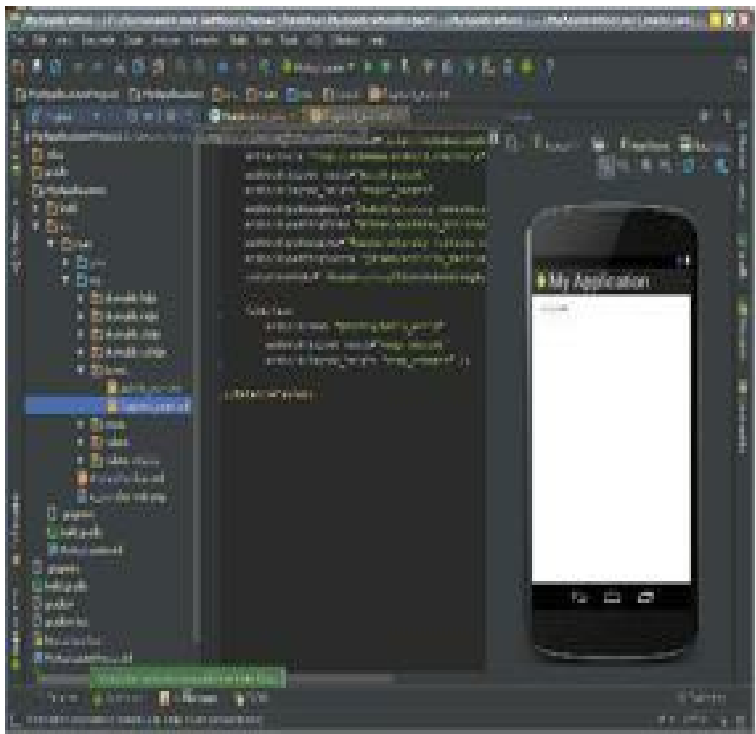


Click on <Finish> to let Android Studio build the project structure. The first time it takes a little bit longer, because Android Studio has to download Gradle (one time).

During the project developing is Internet connection needed. Android Studio needs to connect several times to <http://repo1.maven.com> where the Gradle plugin is hosted.

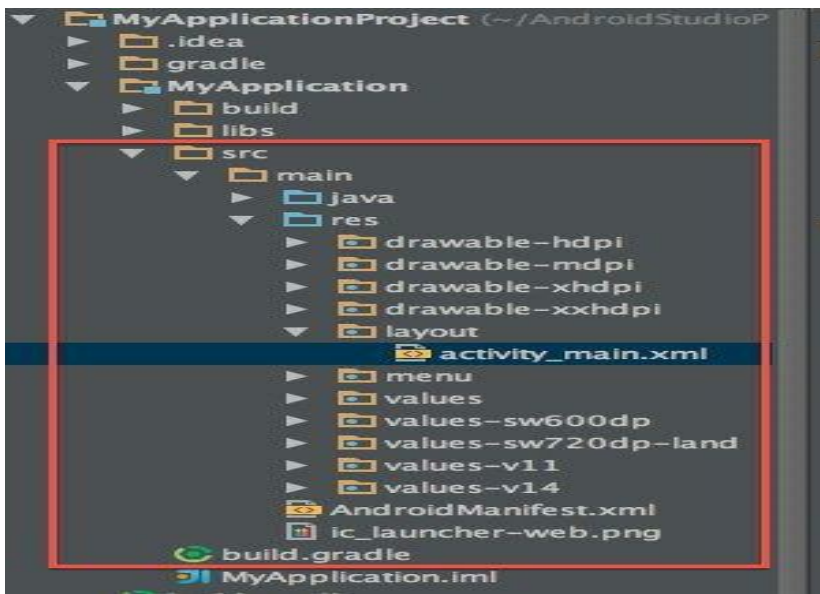
After finishing the building process the project opens.

And this is how it looks like.



You will edit not all of the files you see here. The project files we need to take care of, are marked with red in the next screenshot.

These are the files under src/ plus the build.gradle from the root of the project folder.



Adding the ActionBar to the project

The ActionBar was added to the Android SDK with version 3.0 (API 11). For project like ours, with an API level lower than 11, we can add the ActionBar by using Google's new support library v7 appcompat.

Because we already activated ActionBar in the New Project screen, Android Studio took care of the import of the library and added an entry into the build.gradle file.

To make use of the ActionBar we need to edit the AndroidManifest.xml file and the main.xml in /res/menu/.

The AndroidManifest.xml is located in /src/main/.

Open the AndroidManifest.xml and add the following to your activity:

```
<activity
android:name="com.greendog.androidstudio.example.mynatvieactionbarapplication.MainActivity"
    android:label="@string/app_name"
    android:theme="@style/Theme.AppCompat.Light" >
```

Save and close the AndroidManifest.xml file.

Adding Action Buttons

We're going to add a search button to the ActionBar. To do this, we need the icon for the search button.

Download from this side: <http://developer.android.com/design/downloads/index.html#action-bar-icon-pack>

the Action Bar Icon Pack. Extract the zip file copy the 4 ic_action_search.png from /holo_light/01_core_search into your matching drawable folders.

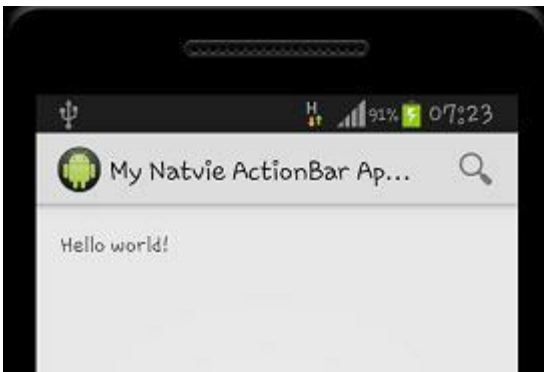
Now open the main.xml file from /res/menu/ and change the content to this:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:greendog="http://schemas.android.com/apk/res-auto" >
    <item
```

```
        android:id="@+id/action_search"
        android:title="@string/action_search"
        android:icon="@drawable/ic_action_search"
        greendog:showAsAction="always" />
<item
    android:id="@+id/action_settings"
    android:title="@string/action_settings"
    greendog:showAsAction="never" />
</menu>
```

The support library v7 appcomt can't use the name space android. For this reason I used my own namespace **greendog**. You can call your namespace whatever you want.

Android Studio 0.3+ is not showing the search icon in preview but on a device. I'm sure that this will change in one of the next updates. The following screenshot shows an app with this configuration running on a device.



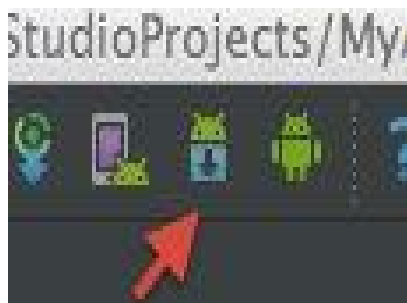
Android SDK Manager

In the Android SDK Manager you can download more Android platforms than the one you got with the installation package. From here you can download the Android Tools, sample, images and some libraries too.

In the following chapters we will come back here.

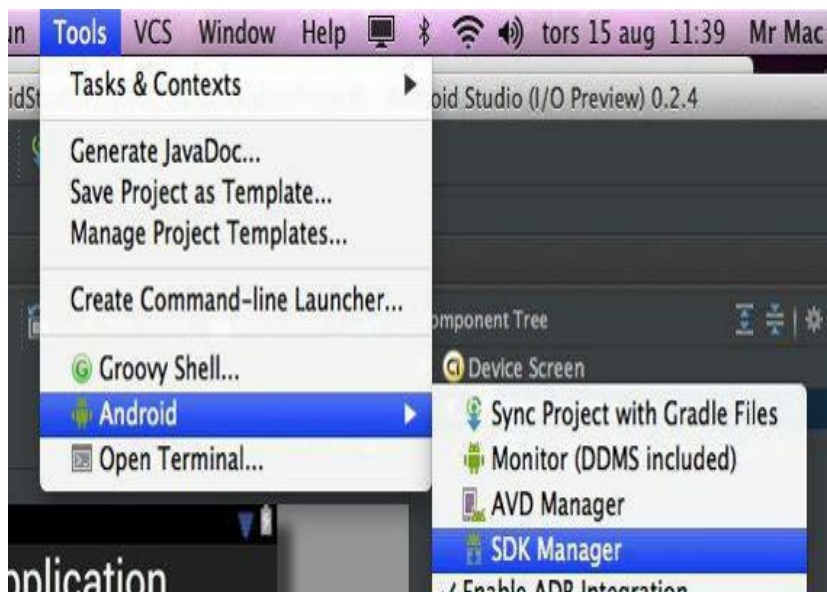
The Android SDK Manager in Android Studio looks and works like in Eclipse.

Call the SDK Manager via the icon



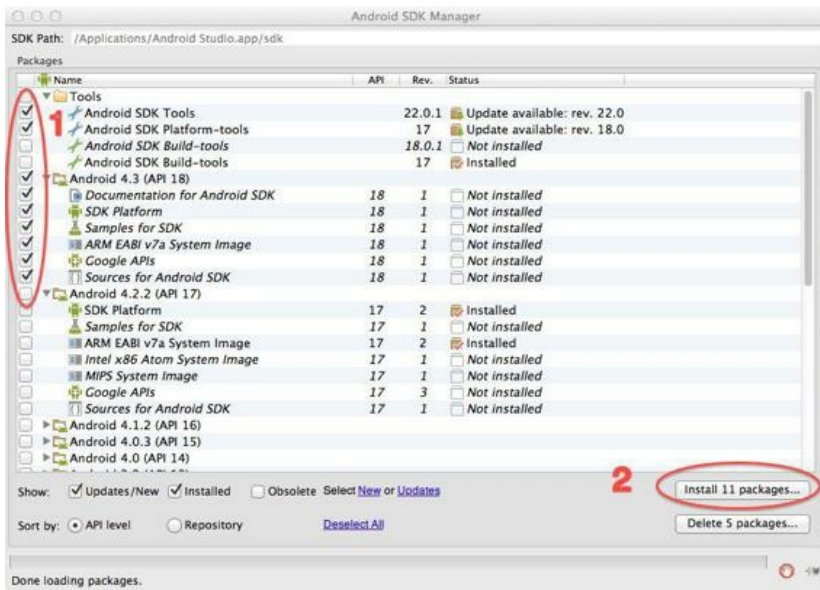
or via

Tools → Android → SDK Manager



Select the needed SDK Platforms and Tools than go to

<Install .. packages..> and click it.

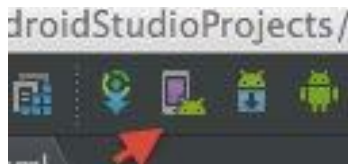


Accept the Licenses and press <Install> to download.

AVD's

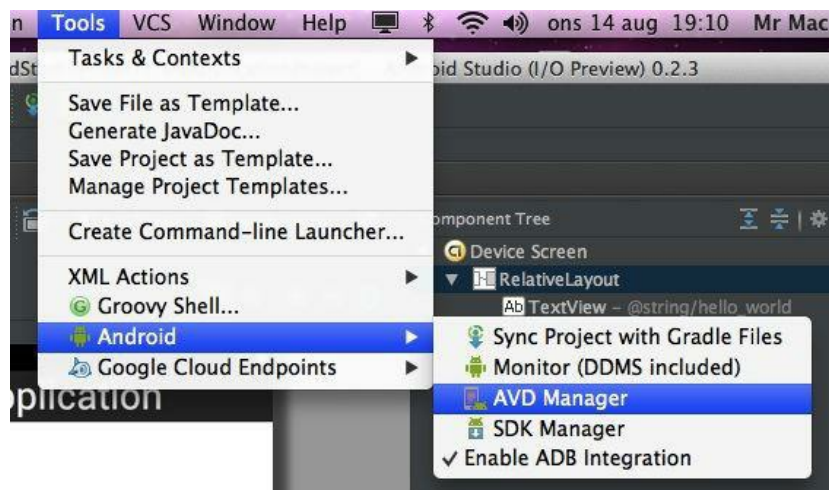
Creating an Emulator or AVD (Android Virtual Devices).

In Android Studio you get to the AVD Manager via the icon

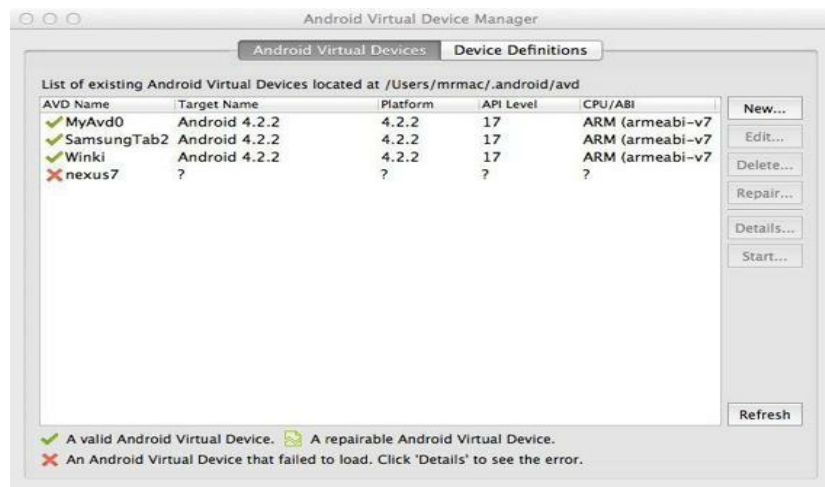


or via

Tools → Android → AVD Manager.



This opens the window "Android Virtual Device Manager".



This is the place where AVD's can be created, edited, repaired or deleted.

And a chosen AVD from the list can be started from here.

Another way to start the Emulator is by click on Run, which gives you the opportunity to choose an Emulator and run the created app on it.

The list of the emulators has a symbol in front of the name. There are three different symbols.

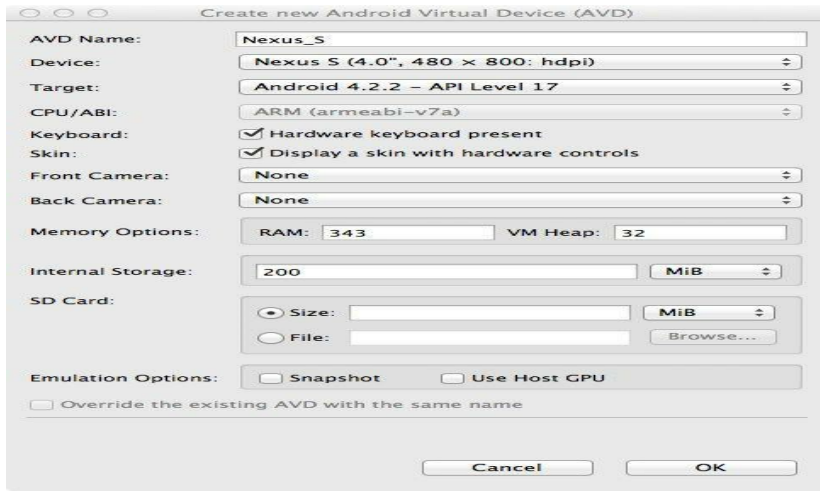
The green hook shows that Emulator has been loaded correctly and can be used for Android Studio projects.

The broken symbol says, the Emulator hasn't been loaded correctly and need to be repaired before it can be used.

The red cross shows that there is something wrong with this Emulator and can't be repaired. This could be for example an Emulator made by Eclipse or IntelliJ until version 12. You can keep these kind of Emulators, there are not bothering Android Studio.

Creating a new Emulator

The "Android Virtual Device Manager" in Android Studio hasn't changed. Everything here works like it works in Eclipse und IntelliJ.



The selection of the <New..> button brings you to the screen "Create new Android Virtual Devices" where you fill in the AVD Name, Device, Android Version, Device platform (intel/arm) and the features the Emulator should have.

AVD Name: you can choose the name freely, but no spaces in the name.

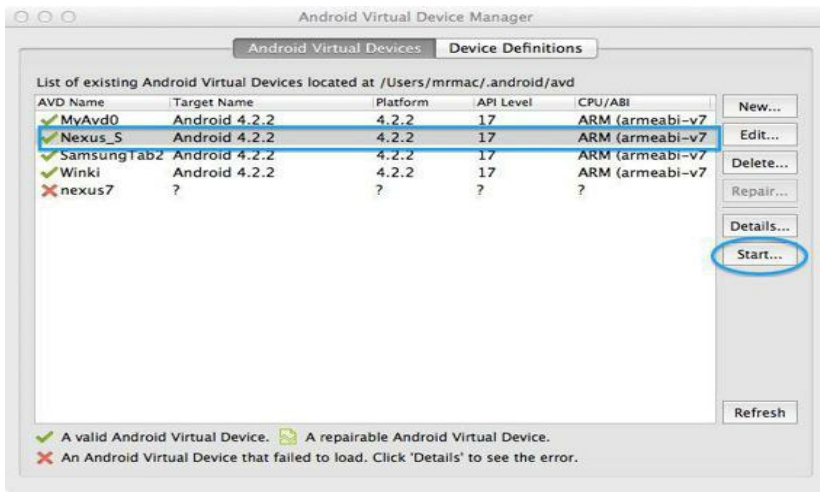
Device: select on from the drop down menu

CPU/ ABI explains a little bit later.

Target: select the target platform from the list. The list depends on the platforms you installed earlier.

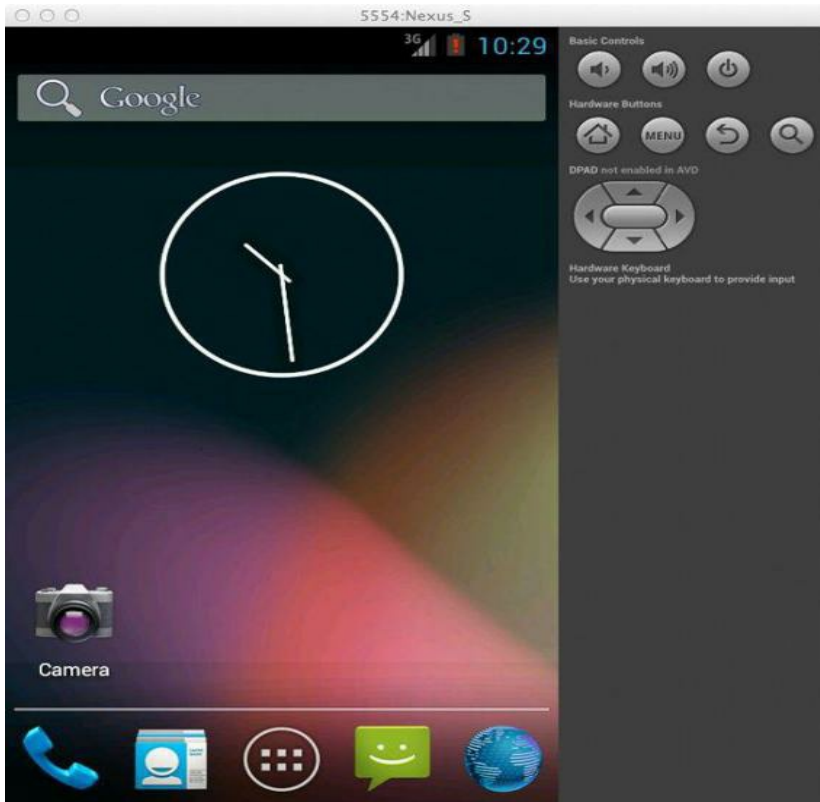
If the app needs an SD card or a camera, select this features.

Click <OK> and a summary of the new Emulator's features are presented. Changes are made via the <Edit> button.



After clicking <Start..> the "Launch Option" window let you choose the screen resolution. This option will show up every time, it is not permanent.

<OK> will start Emulator. This will take some time.



Make the Emulator faster

By choosing some extra settings the Emulator can run up to 50% faster.

Using „Graphics Acceleration“:

This works on Windows, Linux und Mac:

In the screen "Create new Android Virtual Devices" activate „Use host GPU“ (Graphics Acceleration). The API has to be 15 or higher for this.

When you start the Emulator from the command line, it would be look like this:

```
emulator -avd <avd_name>-gpu on
```

Using Intel Atom System Image and Intel x86 Emulator Accelerator (HAXM)

Windows:

1. In the SDK Manager select „Intel x86 Atom System Image for the Android Version you are using (between 15 and 17) and install it.
2. In the SDK Manager under Extras select „Intel x86 Emulator Accelerator (HAXM) and install.
3. Navigate in the File Manager to Extras under the Android Studio directory, than to \intel\Hardware_Accelerated_Execution_Manager. Double-click on „IntelHAXM.exe to install it.
4. In the AVD Manager create a new Emulator and select from CPU/ABI „Intel atom x86“. Set the Ram to 512 or more. Ready to use.

Mac OS X:

The only difference to the Windows Installation is number 3. Here you choose /extras/intel/Hardware_Accelerated_Execution_Manager/IntelHAXM.dmg instead.

Linux:

1. sudo apt-get install kvm
2. In the SDK Manager select „Intel x86 Atom System Image“ for your Android Version (between 15 and 17) and install it.

3. In the AVD Manager create a new Emulator and select from CPU/ABI „Intel atom x86“. Set the Ram to 512 or more. Ready to use.

4. Start the Emulator from the command line with:

```
emulator -avd <avd_name> -qemu -m 512-enable-kvm
```

5. Or put in Build/ Run Configuration under „Additional Emulator Command Line Options“ the following:

```
-qemu -m 512 -enable-kvm
```

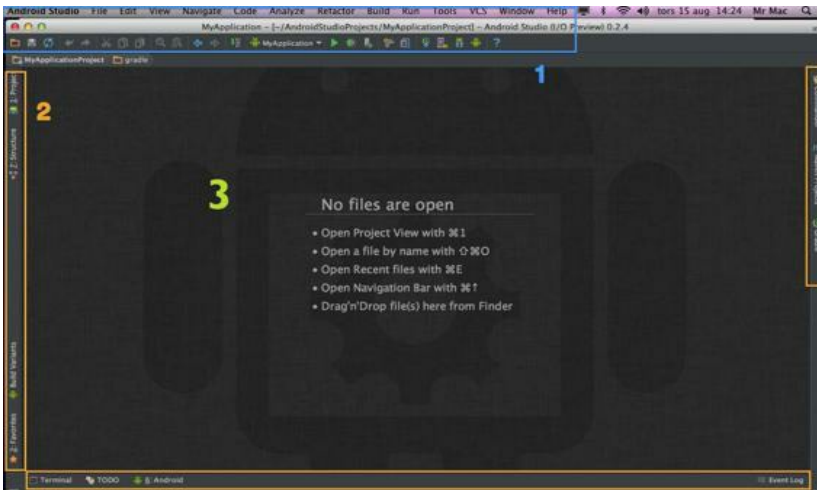
Overview IDE

The workspace

This chapter gives an overview over the most important menu's and symbols to get you started.

After your project is build you are presented with the workspace for your project, but it still looks quite empty. What you can see already is

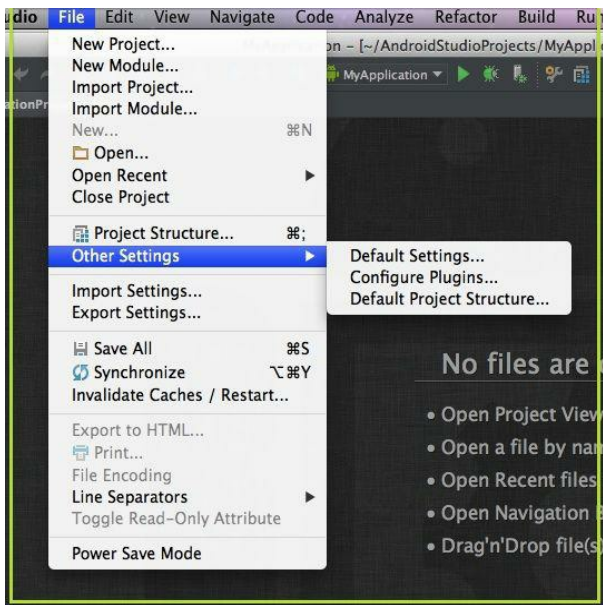
1. The menu list and the toolbar on the top of the workspace. Furthermore you will find here the name of the project, the location of the project files and the version of the Android Studio.
2. on the sides and on the button you will find buttons to open different views, like the button <Project> on the left.
3. the space in the middle, which show right now some tips, will filled up with one or more views when working project files.



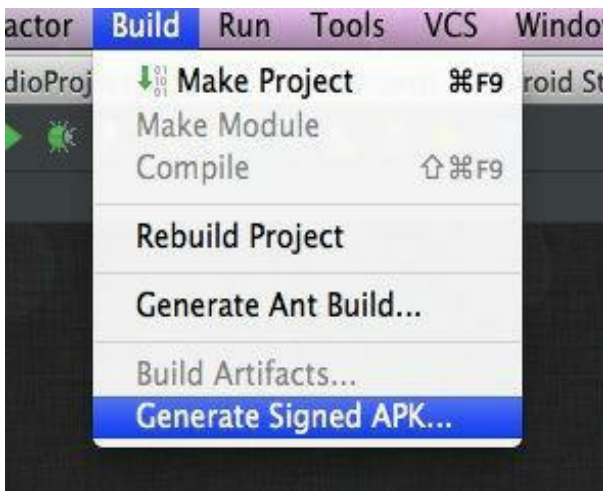
Menu list

Under the File Menu you have all the different configuration settings. If no project is open yet, you can reach the configuration settings from the Welcome Screen via Configure.

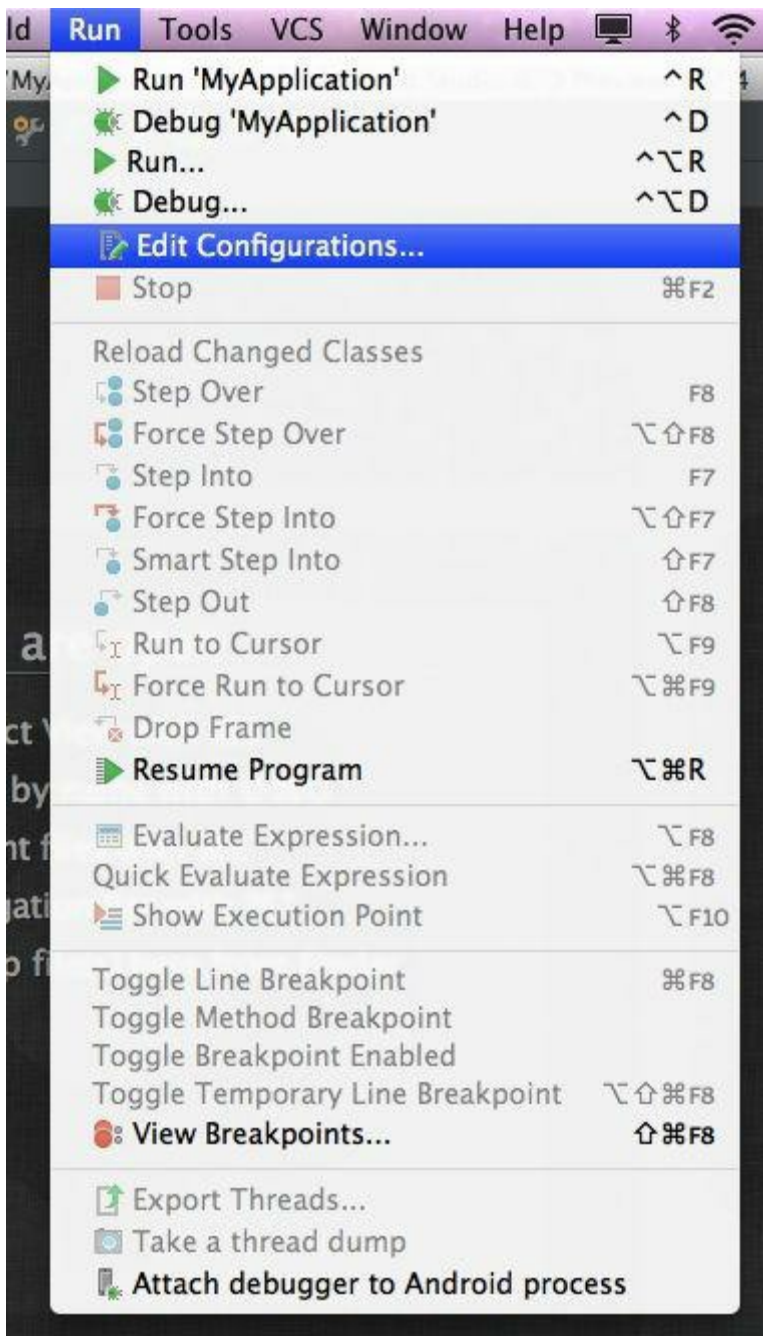
File -> Close Project will close the project and bring you back to the Welcome Screen. To close Android Studio use File -> Exit or on Mac OS X Android Studio -> Quit Android Studio.



From the Build Menu you reach two important functions/ under menus, the Rebuild Project and the Generate Signed APK Menu. Rebuild Project is often useful after copying and configuring a new library into the project. Generate Signed APK is needed to build a signed app for the Android Market.



Run gives you the Run and Build commands and the configuration for the Build and Run in „Edit Configuration“.



Under the Tools menu you will find all the relevant Android menus like AVD Manager or SDK Manager.

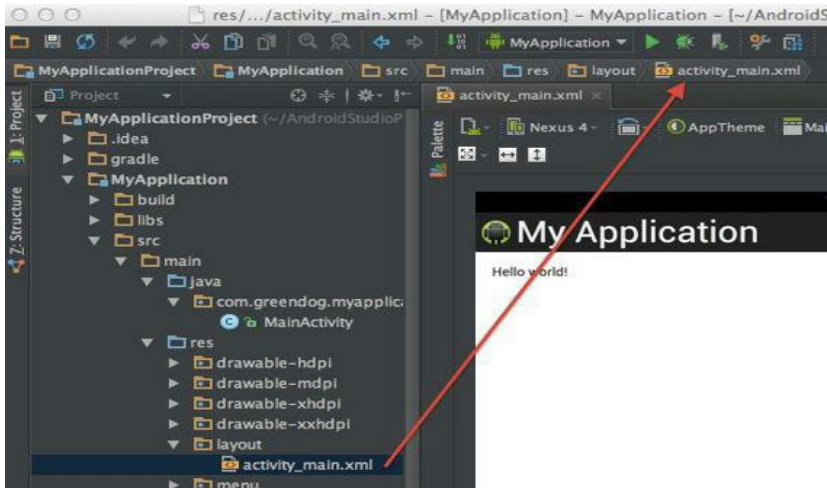
Android part of the symbol list.



Under VCS is the „Version Control Integration“ located. More info about VCS in Chapter 9.

The Help menu is interesting because from here you get all the Keymaps. Many other points in this list are interesting too, but most of them are Internet Links.

Right under the Symbol list you can see the PATH of the file or directory you have marked in the project view. With bigger projects, this can be helpful.

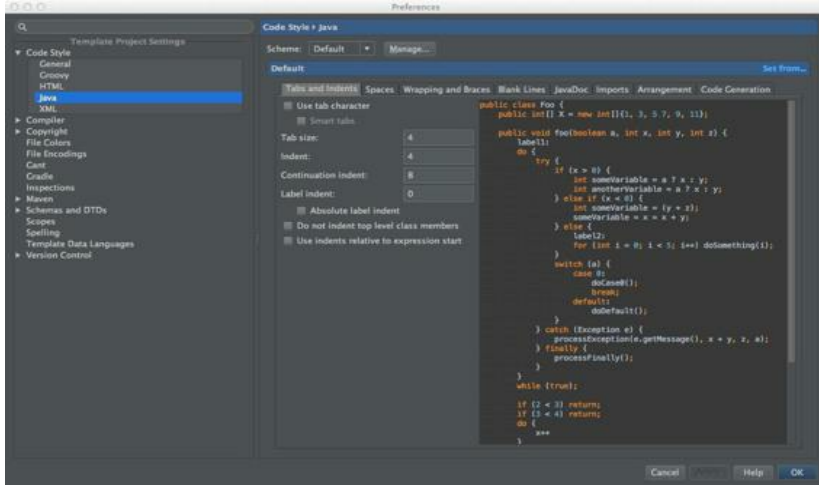


The Editor

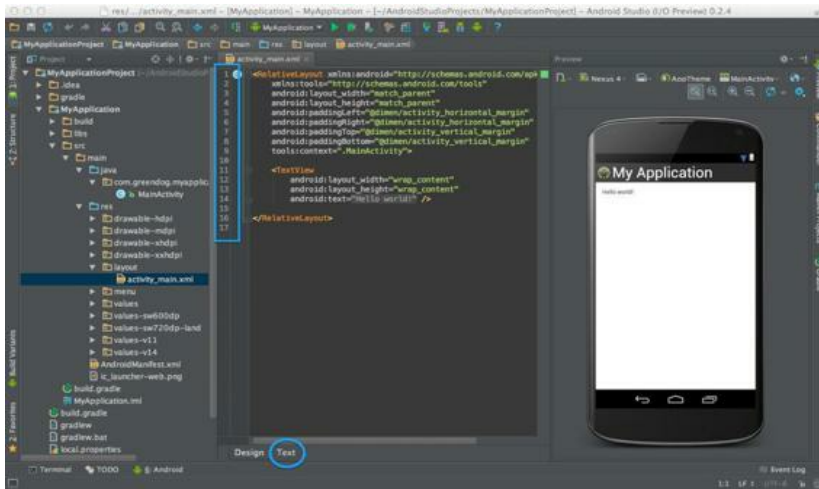
Editor configuration

From File → Other Settings → Default Settings

select "Code Style" to customize the Editor.



Tips and Tricks to the Editor



If you are working on a layout file Text modus, Android Studio gives a live preview in the right panel.

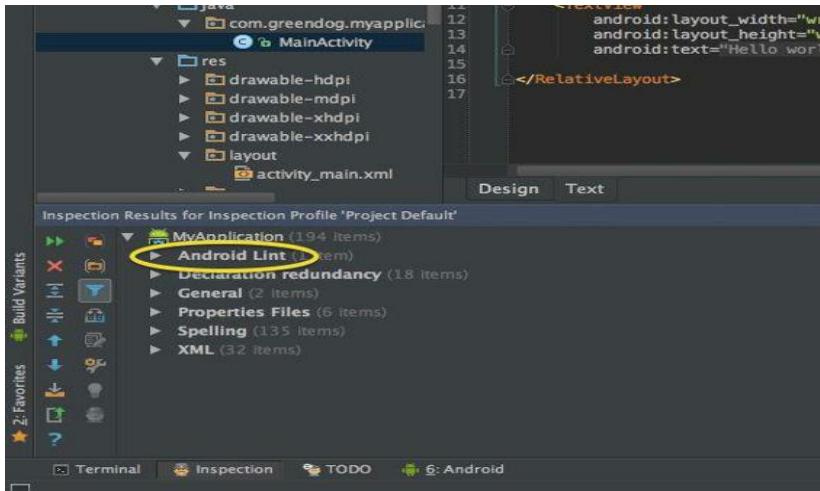
Show Line Numbers

View → Active Editor → Show Line Numbers

Hard coded strings

To discover Java „hard coded strings“ in Java files go to Analyze → Inspect Code

The result can be found in Android Lint → Hard coded text



Renaming

Renaming of variables and constants in a Java file can be done by pointing the mouse on this constant or variable and pressing <Shift><F6>. After renaming, all the dependencies will be updated automatically..

Creating strings on the Fly

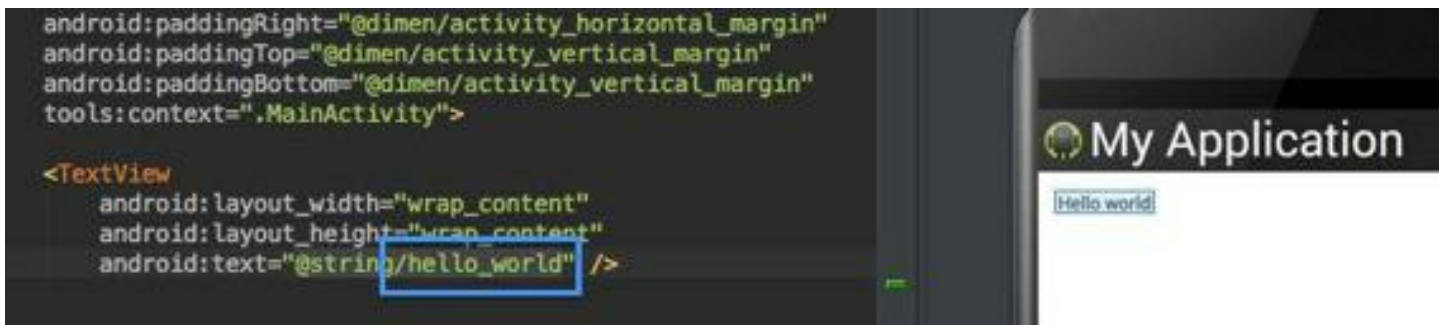
1. Example:

In our example app we like to change the text „HelloWorld“ into „Hello Android World“.

In the opened „activity-main.xml“ file point the mouse on

`android:text="@string/hello_world"`

and change the „*hello_world*“ into „*android_world*“



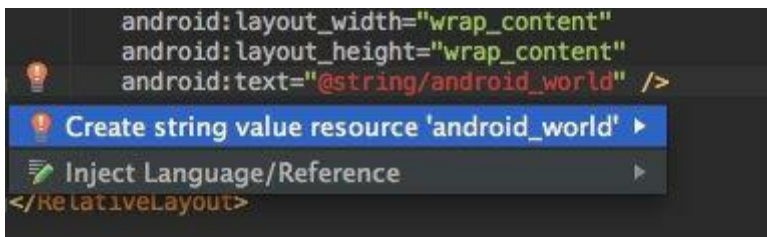
„@string/*android_word*“ is now marked in red color. Which means, Android Studio couldn't find content for this new variable.

```
<TextView
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="@string/android_world" />
```

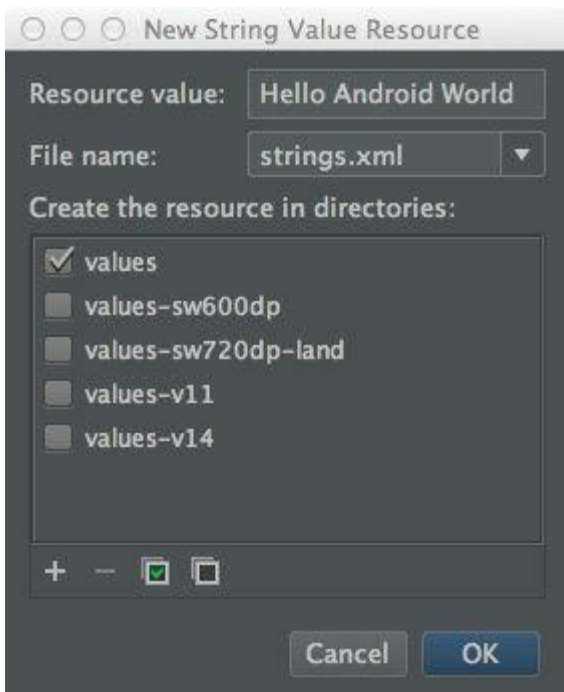
This is correct; we don't have any content for android_world yet.

To change this press <Alt> <Enter>

and choose from the menu „Create string value resource 'android_world'“.



In the next screen we get the chance to write the new value "Hello Android World".



After confirming with <OK> two things happen, the variable name "android_world" changes the color from red to green. And the live preview is showing the new text.



If you only want to change the content of a variable, point with the mouse on it and press

<Apple Key> Mac

<CTRL> Windows

<CTRL> Linux

this opens up the source file in the editor, where the changes can be made.

Changing text size

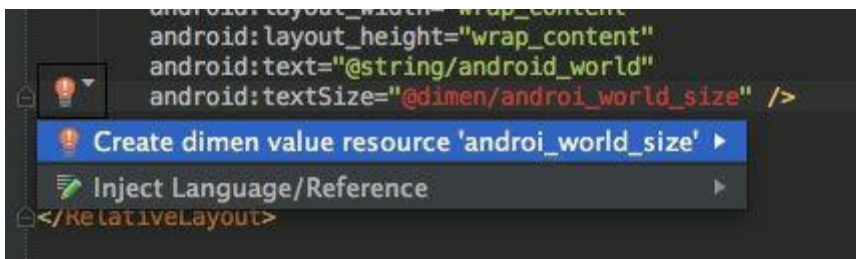
For the moment the text size in our example app is not defined. Therefore Android gives us the standard size for text.

To change the text size, I will use the file `dimens.xml` and make a definition for the new size.

When Android Studio builds a standard project, it creates by default a `dimens.xml` file. So we only have to add a value for our text.

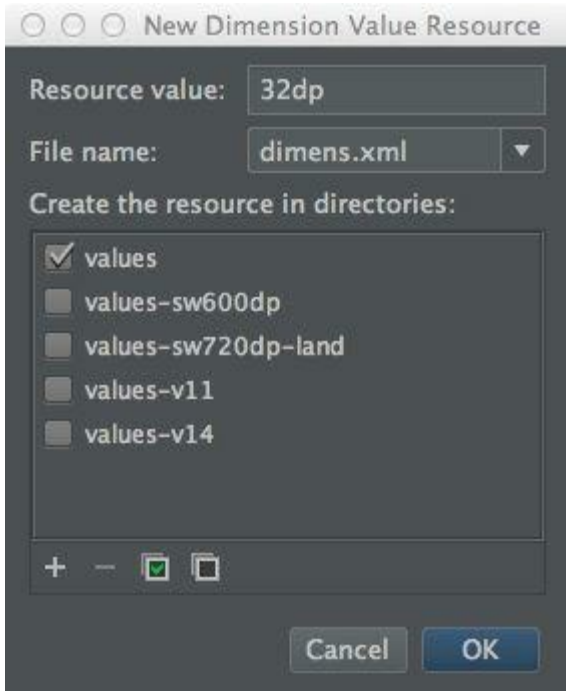
First open the `activity_main.xml` and add

```
android:textSize="@dimen/androi_world_size"
```



By pointing on the new and unresolved variable and pressing <Alt><Enter> the menu shows

„Create dimen value resource 'androi_world_size'“, select it and fill in a value for the dimension.



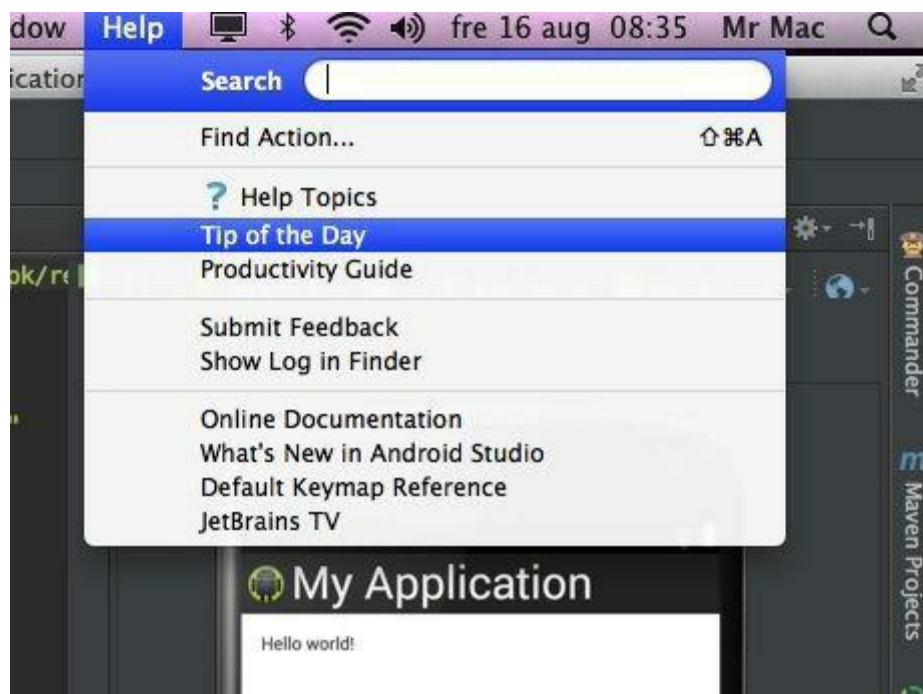
Like the last time, after confirming with <OK> the live preview is showing the changes we made.



More tips for the Editor can be found in „Tips of the Day“.



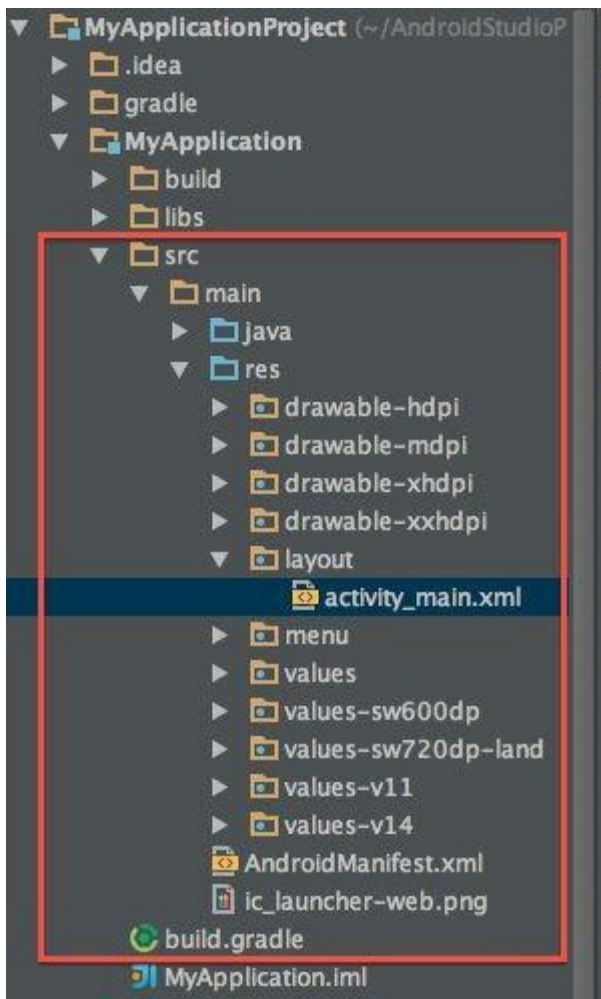
If „Tip of the Day“ is no longer activated, you can reach it all the time from the Help menu.



Project Structure

When you create a new project in Android Studio (or migrate a project from Eclipse), you'll notice that the project structure appears differently than you may be used to. As shown in the screenshot almost all your project files are now inside the `src/` directory, including resources and the manifest file.

The new project structure is due to the switch to a Gradle-based build system. This structure provides more flexibility to the build process and will allow multiple build variants (a feature not yet fully implemented). Everything still behaves as you expect, but some of the files have moved around. For the most part, you should need to modify only the files under the `src/` directory. More information about the Gradle project structure is available in the Gradle Plugin User Guide.



Gradle Build System

What is the Gradle Build System?

Gradle is a Java-based build management and automation tool.

Google integrated Gradle in Android Studio with the help of a newly developed Gradle plugin for Android Studio. The first time an Android project is created, Android Studio will download Gradle 1.7. Therefore, the creation of the first project takes a bit longer, depending on your Internet connection.

In Android Studio the "Gradle build system" is the build system of choice. Gradle replaces "Ant", which is used in Eclipse. "Old Eclipse" projects must be converted from Ant to Gradle before they can be imported to Android Studio. More on this in Chapter 15.

One of the main features of the "Gradle build system" is that you can create different versions of the same app: Free and Paid apps for example.

A script called "build.gradle" largely controls Gradle. The script is written in "Groovy DSL", is easy to read and write.

Build scripts can really be very small, if the project is based on the standard structure. It could look like this:

```
buildscript {  
  
    repositories {  
        mavenCentral()  
    }  
  
    dependencies {  
        classpath 'com.android.tools:gradlexx'  
    }  
  
    apply plugin: 'android'
```

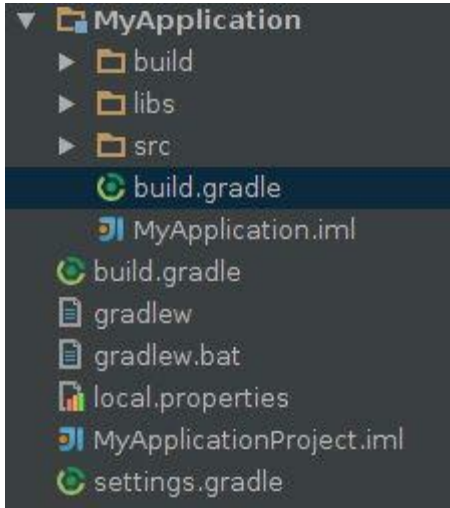
The only real content of the script is "apply plugin: 'android'".

This script will produce an "unsigned apk".

The Standard Gradle File for an Android Project

Which Gradle file is responsible for my project?

The responsible Gradle file is placed in the root of the project folder.



For a single Android app project you only need to customize the build.gradle file. In multi-projects there is additionally the settings.gradle.

The structure of a build.gradle script

By default every new build Android project made by Android Studio get a standard build.gradle file. This file matches the needs for the sample template has. It is perfect for running and building the example project like "My Application".

This is how the build.gradle file looks like for the "My Application" app:

```

buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:0.5.+'
    }
}
apply plugin: 'android'

repositories {
    mavenCentral()
}

dependencies {
    compile 'com.android.support:support-v4:13.0.+'
}

android {
    compileSdkVersion 17
    buildToolsVersion "17.0.0"

    defaultConfig {
        minSdkVersion 8
        targetSdkVersion 16
    }
}

```

The parts of this Gradle file

By default Android Studio the Gradle is connected to the Repository mavenCentral:

Gradle searches in a "repository" to find external libraries. A repository is actually just a collection of files that is organized by group, name and version. Gradle understands various repository formats, such as Maven and Ivy, and different ways to access the repository, such as the local file system or HTTP.

repositories {

mavenCentral()

}

Android Studio needs to use a Gradle plugin to work with Gradle.

dependencies {

classpath 'com.android.tools.build:gradle:0.5.+'

```
}
```

This plugin is telling us that this Android project is an application.

```
apply plugin: 'android'
```

If your Android project is a library project, you need to change the plugin into plugin: 'android-library'.

The template we used for this example app has the android-support-v4 library already integrated.

```
dependencies {
```

```
compile 'com.android.support:support-v4:13.0.+'
```

```
}
```

These informations have been collected during the creation process of the app.

```
android {
```

```
compileSdkVersion 17
```

```
buildToolsVersion "17.0.0"
```

```
defaultConfig {
```

```
minSdkVersion 8
```

```
targetSdkVersion 16
```

```
}
```

```
buildToolsVersion "17.0.0"
```

```
defaultConfig {
```

```
minSdkVersion 8
```

```
targetSdkVersion 16
```

```
}
```

}

The Gradle-Skript has 3 major parts:

1. buildscript { ... }

configures the code driving the build.

In this case, this declares that it uses the Maven Central repository, and that there is a classpath dependency on a Maven artifact. This artifact is the library that contains the Android plugin for Gradle in version 0.5.6

Note: This only affects the code running the build, not the project. The project itself needs to declare its own repositories and dependencies. This will be covered later.

2. The 'android' plugin,

The plugin provides everything to build and test Android applications.

3. android { ... }

android { ... } configures all the parameters for the android build. This is the entry point for the Android DSL.

By default, only the compilation target is needed. This is done with the compileSdkVersion property.

This is the same as the target property in the project.properties file of the old build system. This new property can either be assigned an int (the api level) or a string with the same value as the previous target property.

Has the build.gradle file to be edited before building or running a project?

No, if your project does not use more libraries than the android-support-v4 library and you don't need different version of your app .. properly not.

Examples, when you need customized the build.gradle file:

- when using ActionBar Sherlock
- when using Admob or other Ads
- with Test Projects

- with Multi-app Projects ...

The Gradle file(s) can be edited using the Android Studio editor. Double-click the build.gradle file from the Project View window on the left to open it in the editor.

Using Gradle from the IDE or from the command line

To build and run the project from the IDE, click Build or Run from the Menu.

To use Gradle from the command line, open a terminal window and navigate into the project folder.

Example:

The AndroidStudioProjects folder is the folder where all our Android Studio projects are getting stored.

After creating the example "My Application", we got a new folder under AndroidStudioProjects called MyApplicationProject. Navigate into this folder. Here you will find the Gradle Wrapper command line tool. For Windows use the gradlew.bat. For Linux und Mac OS X use gradlew.

The command

./gradlew assemble

gradlew.bat assemble

will build 2 files, a debug file and a release file.

Das Android plugin has 6 "anchor tasks":

assemble

The task to assemble the output(s) of the project

check

The task to run all the checks.

connectedCheck

Runs a check that requires a connected device or emulator. they will run on all connected devices in parallel.

deviceCheck

Runs checks using APIs to connect to remote devices. This is used on CI servers.

build

This task does both assemble and check

clean

This task cleans the output of the project

From the command line you can get the high level task running the following command:

```
./gradlew tasks
```

```
gradlew.bat tasks
```

For a full list and seeing dependencies between the tasks run:

```
./gradlew tasks --all
```

```
gradlew tasks --all
```

Fixing Gradle after update to Android Studio 0.3+

I had problems by myself and it turned out, I had to update the gradle plugin and Gradle.

This is what I've done with a project that was created with Android Studio 0.2.13

I updated the gradle plugin in build.gradle to 0.6+

```
dependencies {  
    classpath 'com.android.tools.build:gradle:0.6.+'  
}
```

But when using the gradle plugin 0.6+, Android Studio needs Gradle 1.8. So, you have to update Gradle from 1.7 to 1.8.

This was done by going to

Tools -> Android -> Synch Project with Gradle Files

It forces Android Studio to give out Error message you see on <http://tools.android.com/recent>

In the error message you have to click the part "Fix Gradle wrapper and re-import project". I restarted, not sure if this make a difference.

So far everything seems to work. I can create new projects and it build fast, without errors, but the gradle has still 0.5+.

Looks like, I had to change it manually for every new projects. This is what I thought, because I didn't get any error or information from this installation to fix it. (Mac).

Than I went to my Windows installation, which had still Android Studio version 0.2.13 and

tried the same procedure, I used with the updated version, changed the build,gradle and used Synch Project with Gradle Files. This gave me a total different information:

Gradle version 1.8 is required. Current version is 1.7. If using the gradle wrapper, try editing the distributionUrl in

C:\Documents and Settings\Owner\Start Menu\Programs\Android Studio\gradle\wrapper\gradle-wrapper.properties to gradle-1.8-all.zip

Turned out, I have a gradle-wrapper.properties file in every project (in gradle/) but I don't have any gradle-wrapper.properties file at the location Android Studio gave me.

When you open the gradle-wrapper.properties file from your project, it will properly look like this:

```
#Sun Oct 20 03:35:31 CEST 2013
distributionBase=GRADLE_USER_HOME
distributionPath=wrapper/dists
zipStoreBase=GRADLE_USER_HOME
zipStorePath=wrapper/dists
distributionUrl=http://services.gradle.org/distributions/gradle-1.7-
bin.zip
```

To change the gradle-1.7-bin.zip into gradle-1.8-rc-1-bin is possible but not the solution I was looking for.

Check out if Andoid Studio already downloaded the gradle plugin 1.8

Directory of C:\Documents and Settings\Owner\gradle\wrapper\dists

```
2013-10-13 06:24 <DIR> .
2013-10-13 06:24 <DIR> ..
2013-10-04 18:17 <DIR> gradle-1.6-bin
2013-10-13 06:24 <DIR> gradle-1.7-all
2013-10-04 12:23 <DIR> gradle-1.7-bin
2013-10-04 18:26 <DIR> gradle-1.8-rc-1-bin
0 File(s)      0 bytes
```

6 Dir(s) 81 945 972 736 bytes free

Now I create a new gradle-wrapper.properties file in

C:\Documents and Settings\Owner\Start Menu\Programs\Android Studio\gradle\wrapper\

```
#Tue Oct 20 06:35:40 CEST 2013
```

```
distributionBase=GRADLE_USER_HOME
```

```
distributionPath=wrapper/dists
```

```
zipStoreBase=GRADLE_USER_HOME
```

```
zipStorePath=wrapper/dists
```

```
distributionUrl=http://services.gradle.org/distributions/gradle-1.8-all.zip
```

This did the Trick. Now, after updating to 0.3, every new project gets a build.gradle with dependencies {

```
    classpath 'com.android.tools.build:gradle:0.6.+'
```

```
}
```

and a gradle-wrapper.properties file in gradle/ with

```
distributionUrl=http://services.gradle.org/distributions/gradle-1.8-bin.zip
```

If you prefer a clean Android Studio 0.3.1 installation go to:

<http://tools.android.com/download/studio/canary/latest>

Libraries and dependencies

Libraries come in different versions: as

1. single Java file / jar file example: support-v4: 13.0
2. Java library project directory: Example ActionBar Sherlock

Incorporate a "remote jar" library

In the sample project, the android-support-v4 library is automatically included and is coming from the Maven repository. Therefore 2 entries are needed to integrate the library.

The first entry is

```
repositories {  
  
    mavenCentral()  
}
```

The second entry is

```
dependencies {  
  
    compile 'com.android.support:support-v4:13.0.+'  
}
```

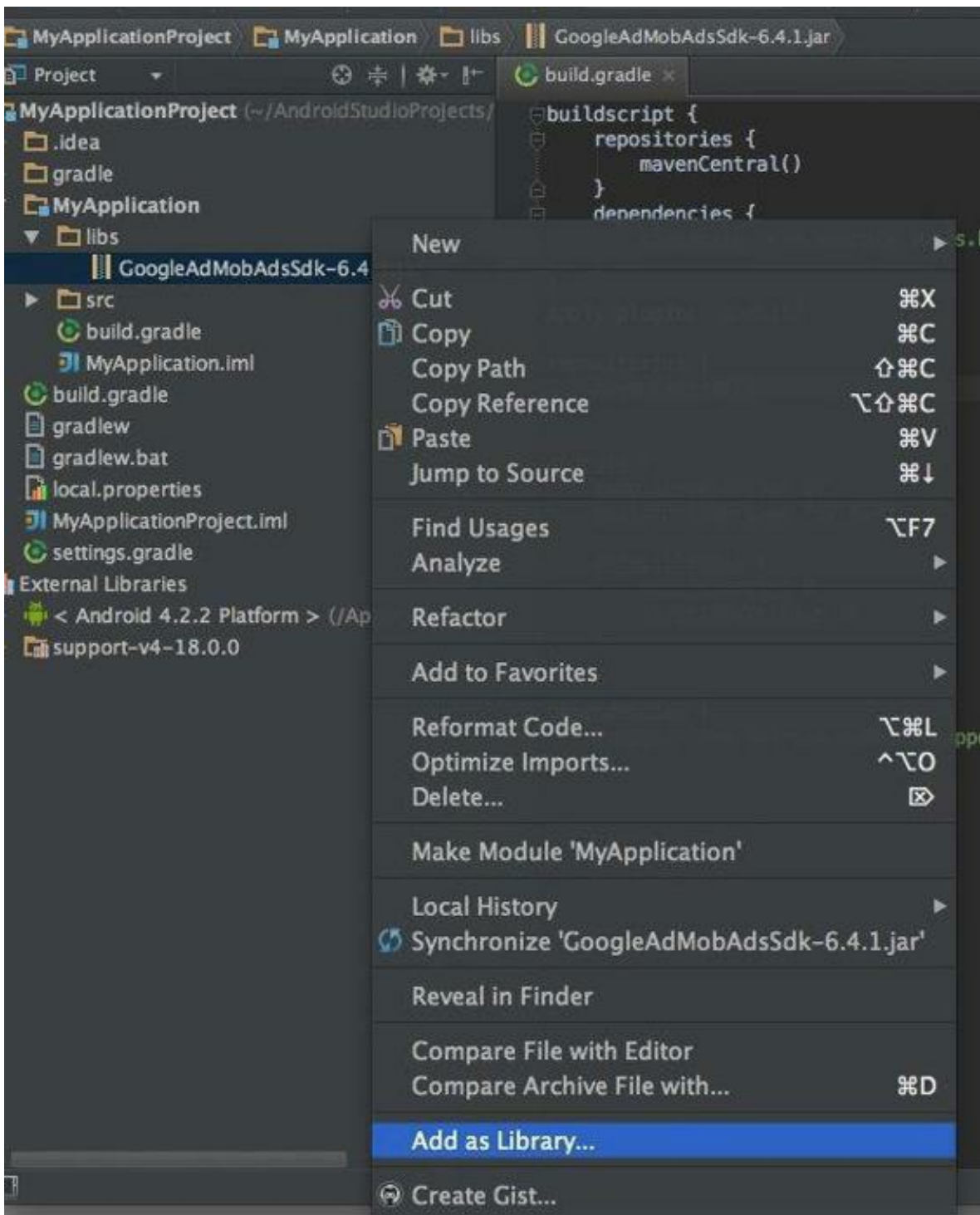
the name of the library.

```
dependencies {  
    compile 'com.android.support:support-v4:18.0.0'  
}
```

Other extern libraries are integrated in the same way.

Integration of a local library

The local library must be located in a directory of this project. First, create a directory on the height of the src / directory in which the library is to be copied later. Than copy or drag the library into this folder.



Click with the right mouse on the library and select „Add as Library“ from the context menu.

In the build.gradle file has to be add the entry

```
compile files('libs/android-support-v4.jar')
```

Example GoogleAdMobAdsSdk

1. Download GoogleAdMobAdsSdk-6.4.1.jar

<https://developers.google.com/mobile-ads-sdk/download>

and put the unpacked file in the libs/ directory of your project

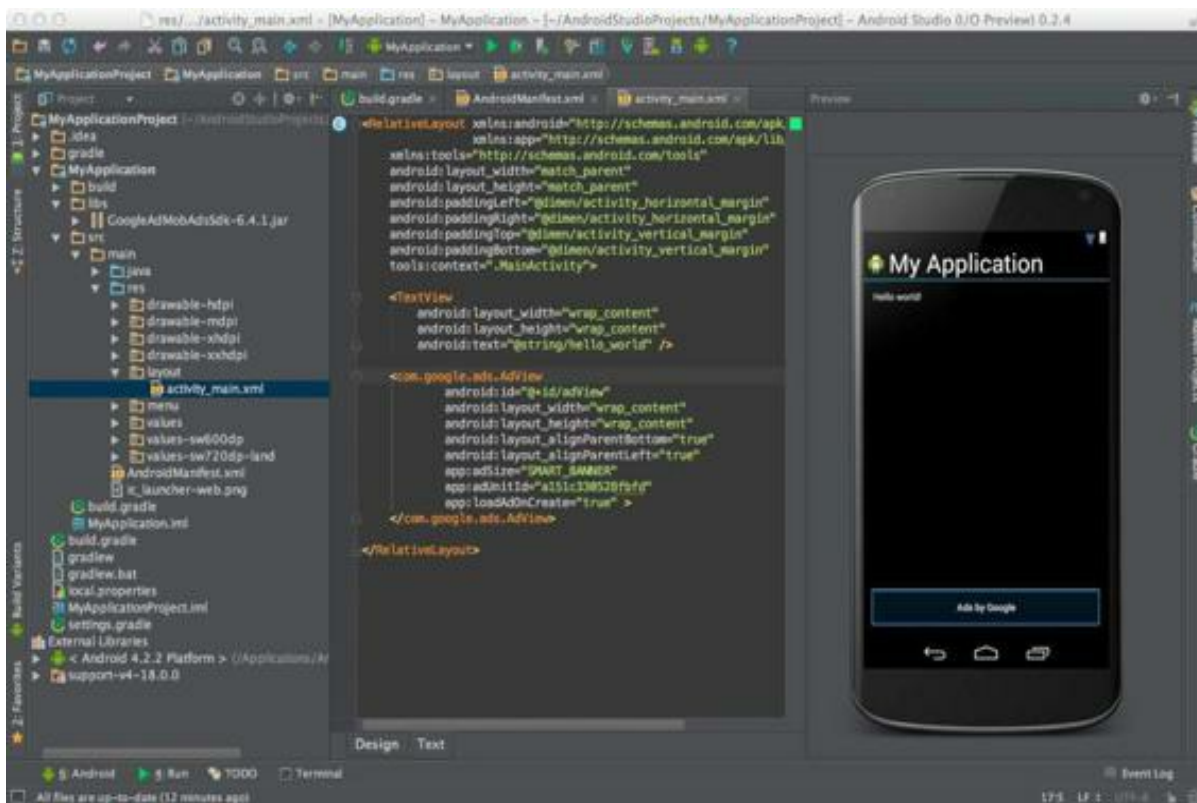
2. Click with the right mouse on the library and select „Add as Library“(Global)

3. Open the build.gradle and add

```
dependencies {  
    compile files('libs/android-support-v4.jar', 'libs/GoogleAdMobAdsSdk-6.4.1.jar')  
}
```

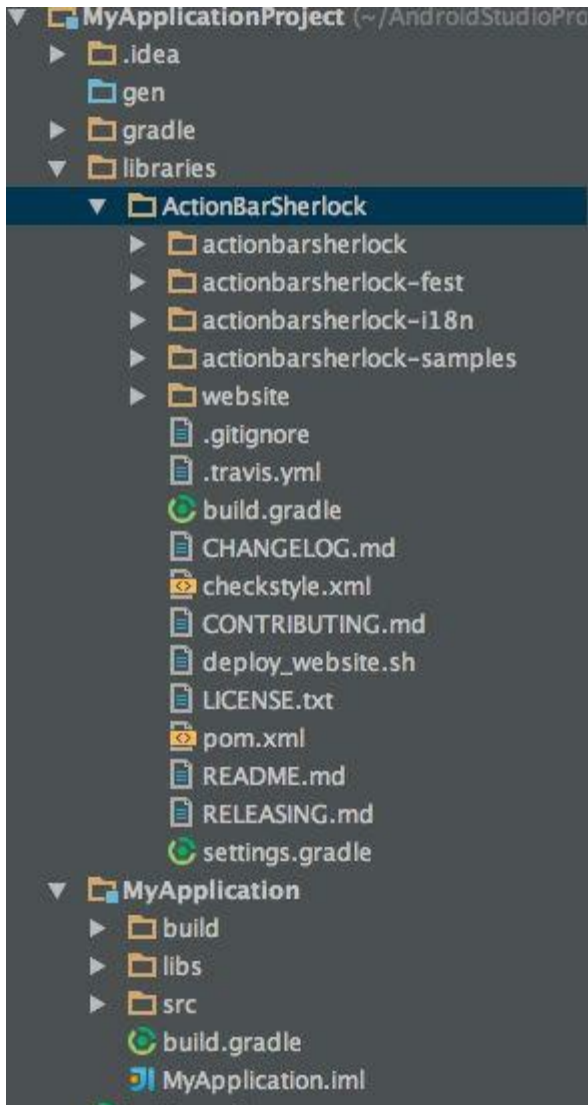
4. Menu Build -> Rebuild Project

5. The Admob library is ready to use and can now be imported into Java files, in layout files can Ads be added. Finally, the AndroidManifest need uses-permission for Internet, the AdActivity and the Admob ID you got for your app.



Example: ActionBar Sherlock

1. Download the newest ActionBar Sherlock with Gradle
2. `git clone git://github.com/JakeWharton/ActionBarSherlock.git`
3. Put the library folder into your project structure like this.



4. Open the **settings.gradle** and add the part for the ActionBar Sherlock.

```
include ':MyApplication', ':ActionBarSherlock:actionbarsherlock'
```

```
include ':MyApplication', ':libraries:ActionBarSherlock:actionbarsherlock'
```

The recommendation from ActionBar Sherlock is, to use the .aar library. This is explained in chapter "Building an .aar library file".

Local installation of Gradle 1.7

There can be several reasons why it's not enough to use Android Studio's Gradle.

For example, if you like to build a local Maven repository or to build .aar library files.

This is what you need to do:

Download the *gradle-1.7-all.zip* package for all OS from here:

<http://www.gradle.org/downloads>

For running Gradle, add `GRADLE_HOME/bin` to your `PATH` environment variable. Usually, this is sufficient to run Gradle.

For Linux and Mac OS X users

Unzip the downloaded file and maybe rename the folder into *gradle*, like me, or use in the following commands your folder name instead of *gradle*.

```
sudo cp -R gradle /usr/local
export GRADLE_HOME=/usr/local/gradle
export GRADLE_HOME=$GRADLE_HOME/bin/
export PATH=/usr/local/gradle/bin:$PATH
```

Test with the command `gradle -v` if everything is working.

As a result, you should see something like this:

```
-----
Gradle 1.7
-----
```

```
Build time: 2013-08-06 11:19:56 UTC
```

```
Build number: none
```

```
Revision: 9a7199efaf72c620b33f9767874f0ebced135d83
```

```
Groovy: 1.8.6
```

Ant: Apache Ant(TM) version 1.8.4 compiled on May 22 2012

Ivy: 2.2.0

JVM: 1.7.0_21 (Oracle Corporation 23.21-b01)

OS: Linux 3.2.0-44-generic i386

Keep in mind that you use the *gradle* command here and not *gradlew* when working with Android Studio projects.

Local installation of Maven 2

A local installation of Maven is necessary when you need to build a local Maven repository. For example to put the ActionBarSherlock .aar file in there.

And because Android Studio is using Maven 2, I stay with Maven 2 for the local installation.

First download the package *apache-maven-2.2.1-bin.tar.gz* from:

<http://maven.apache.org/download.cgi>

Set the `MAVEN_HOME` environment variable to the path where you extracted maven, and add the maven's bin folder to the `PATH` environment variable.

For Windows users

1. Unzip the distribution archive, i.e. *apache-maven-3.1.0-bin.zip* to the directory you wish to install Maven 3.1.0. These instructions assume you chose `C:\Program Files\Apache Software Foundation`. The subdirectory *apache-maven-3.1.0* will be created from the archive.
2. Add the `M2_HOME` environment variable by opening up the system properties (WinKey + Pause), selecting the "Advanced" tab, and the "Environment Variables" button, and then adding the `M2_HOME` variable in the user variables with the value `C:\Program Files\Apache Software Foundation\apache-maven-3.1.0`. Be sure to omit any quotation marks around the path even if it contains spaces. Note For Maven 2.0.9: make sure that the `M2_HOME` doesn't have a `\` as last character.
3. In the same dialog, add the `M2` environment variable in the user variables with the value `%M2_HOME%\bin`.
4. Optional: In the same dialog, add the `MAVEN_OPTS` environment variable in the user variables to specify JVM properties, e.g. the value `-Xms256m -Xmx512m`. This environment variable can be used to supply extra options to Maven.
5. In the same dialog, update/create the `Path` environment variable in the user variables and prepend the value `%M2%` to add Maven available in the command line.
6. In the same dialog, make sure that `JAVA_HOME` exists in your user variables or in the system variables and it is set to the location of your JDK, e.g. `C:\Program Files\Java\jdk1.5.0_02` and that `%JAVA_HOME%\bin` is in your `Path` environment variable.
7. Open a new command prompt (Winkey + R then type `cmd`) and run `mvn --version` to verify that it is correctly installed.

For Linux and Mac OS X users

1. Extract the distribution archive, i.e. `apache-maven-3.1.0-bin.tar.gz` to the directory you wish to install Maven 3.1.0. These instructions assume you chose `/usr/local/apache-maven`. The subdirectory `apache-maven-3.1.0` will be created from the archive.
2. In a command terminal, add the `M2_HOME` environment variable, e.g. `export M2_HOME=/usr/local/apache-maven/apache-maven-3.1.0`.
3. Add the M2 environment variable, e.g. `export M2=$M2_HOME/bin`.
4. Optional: Add the `MAVEN_OPTS` environment variable to specify JVM properties, e.g. `export MAVEN_OPTS="-Xms256m -Xmx512m"`. This environment variable can be used to supply extra options to Maven.
5. Add M2 environment variable to your path, e.g. `export PATH=$M2:$PATH`.
6. Make sure that `JAVA_HOME` is set to the location of your JDK, e.g. `export JAVA_HOME=/usr/java/jdk1.5.0_02` and that `$JAVA_HOME/bin` is in your `PATH` environment variable.
7. Run `mvn --version` to verify that it is correctly installed.

The result of the test should be look like this:

Apache Maven 2.2.1 (rdebian-8)

Java version: 1.6.0_45

Java home: `/usr/lib/jvm/java-6-oracle/jre`

Default locale: `en_US`, platform encoding: `UTF-8`

OS name: "linux" version: "3.2.0-44-generic" arch: "i386" Family: "unix"

Building an .aar library file

I assume that you have JDK, Maven and Gradle 1.7 installed.

For this example I put the ActionBarSherlock project library on my GitHub. Download the ActionBarSherlock project library and if it is zipped, unpack it. This is the version 4.4.0.

Now in Android Studio, select from the Welcome Screen -> Import project

and navigate to the downloaded *ActionBarSherlock* folder, go to the underlying folder *actionbarsherlock* and click on the *build.gradle* file to import the library project.

From the command line navigate into the *actionbarsherlock* folder and use your local Gradle installation to build.

```
gradle assemble
```

this gives you a new folder inside of *actionbarsherlock* with the name build, and the .aar files are under *libs/*.

Making a local Maven repository from the actionbarsherlock-4.4.0.aar

From the command line:

```
mvn install:install-file -Dfile=/path_to_your_lib/actionbarsherlock-4.4.0.aar -DgroupId=com.actionbarsherlock -DartifactId=actionbarsherlock -Dversion=4.4.0 -Dpackaging=aar
```

The result should look like this:

```
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'install'.
[INFO] -----
-----
[INFO] Building Maven Default Project
[INFO] task-segment: [install:install-file] (aggregator-style)
[INFO] -----
-----
[INFO] [install:install-file {execution: default-cli}]
[INFO] Installing /home/janebabra/actionbarsherlock.aar to /home/janebabra/
{version}.aar
[INFO] -----
-----
[INFO] BUILD SUCCESSFUL
[INFO] -----
-----
[INFO] Total time: 1 second
[INFO] Finished at: Wed Sep 04 13:52:01 CEST 2013
[INFO] Final Memory: 6M/15M
[INFO] -----
-----
```

Using the .aar library in your Android application project

Open the Project View and add to your root directory a file named *local.properties* with the following content:

```
sdk.dir=/path_to_android_sdk/android-studio/sdk
```

Make the changes in the build.gradle file and you are done.

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:0.5.+'
    }
}
apply plugin: 'android'

repositories {
    mavenCentral()
    mavenLocal()
}

dependencies {
    compile('com.example:actionbarsherlock:4.4.0')
}

android {
    compileSdkVersion 17
    buildToolsVersion "17.0.0"

    defaultConfig {
        minSdkVersion 7
        targetSdkVersion 17
    }
}
```

Import ActionBarSherlock like this.

```
package com.example.myabsaarproject;

import android.os.Bundle;
import com.actionbarsherlock.view.Menu;
import com.actionbarsherlock.app.SherlockActivity;

public class MainActivity extends SherlockActivity {
```

5. Open the **build.gradle** file from the ActionBarSherlock folder and add the missing pieces:

```
buildscript {
    repositories {
        maven { url 'http://repo1.maven.org/maven2' }
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:0.5.+'
    }
}
apply plugin: 'android-library'

repositories {
    mavenCentral();
}

dependencies {
    compile files('libs/android-support-v4.jar')
}

android {
    compileSdkVersion 17
    buildToolsVersion "17.0.0"

    defaultConfig {
        minSdkVersion 7
        targetSdkVersion 16
    }

    sourceSets {
        main {
            manifest.srcFile 'AndroidManifest.xml'
            java.srcDirs = ['src']
            res.srcDirs = ['res']
        }
    }
}
```

Open the **build.gradle** for the project and add the following:

```
dependencies {
    compile project(':ActionBarSherlock:actionbarsherlock')
}
```

The build.gradle file from the project must not have the dependencies for the android-support-v4 library

In the final release a project library like this should be integrate as ".aar" file, similar to the

inclusion of "jar" files. Once this works, the chapter will be updated.

Version Control (VCS)

Android Studio supports the following VCS: **Git, GitHub, Mercurial and Subversion.**

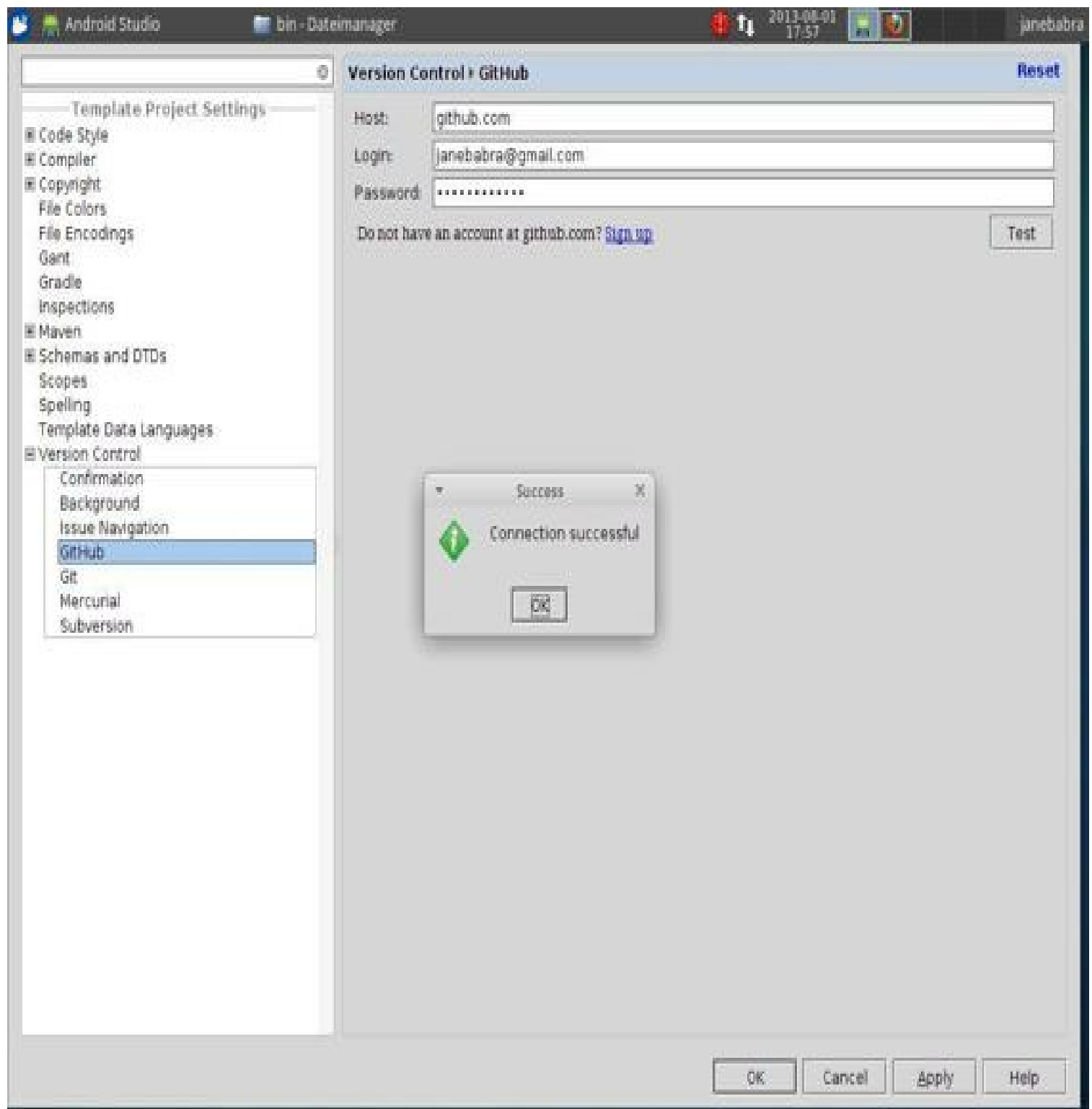
To use one of the systems, the appropriate program that is used by the VCS has to be installed on the computer. For Git and GitHub would this be "git" for Mercurial "hg" and subversion "svn".

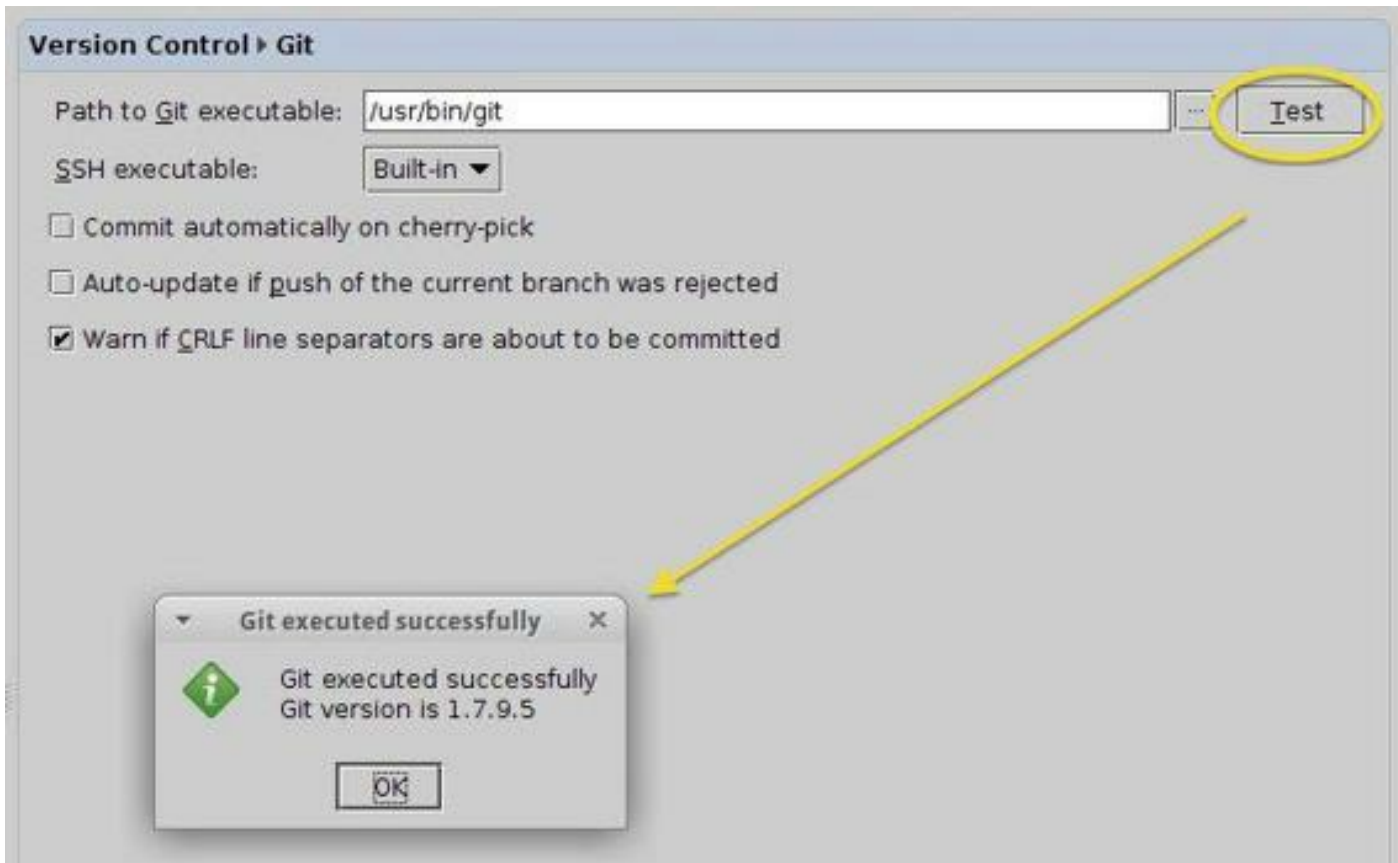
Configuring VCS for your project

From the Welcome Screen -> Project Defaults -> Settings -> Version Control

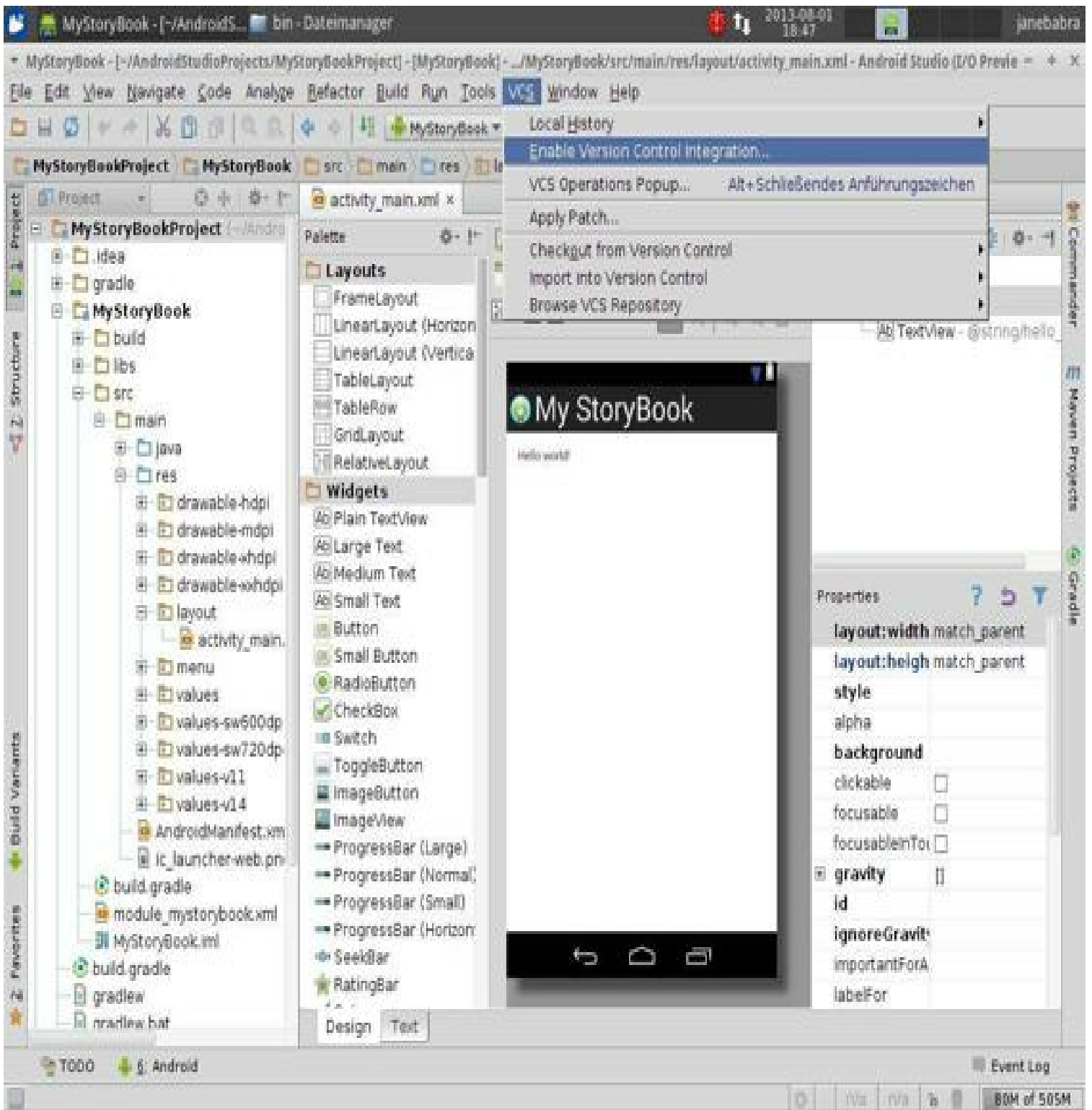
GitHub

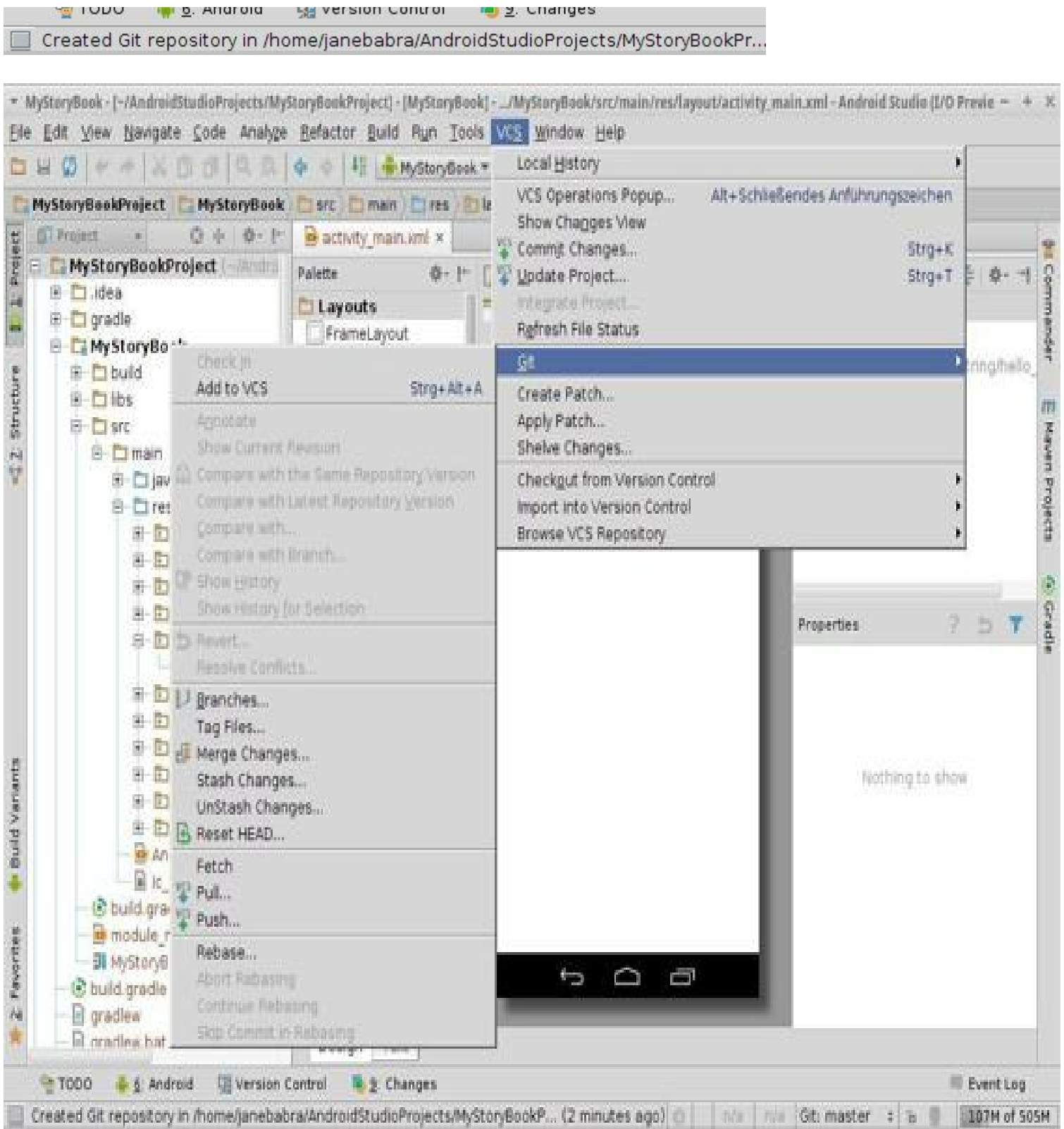
Write down your login information from your GitHub account or click on Sign up.



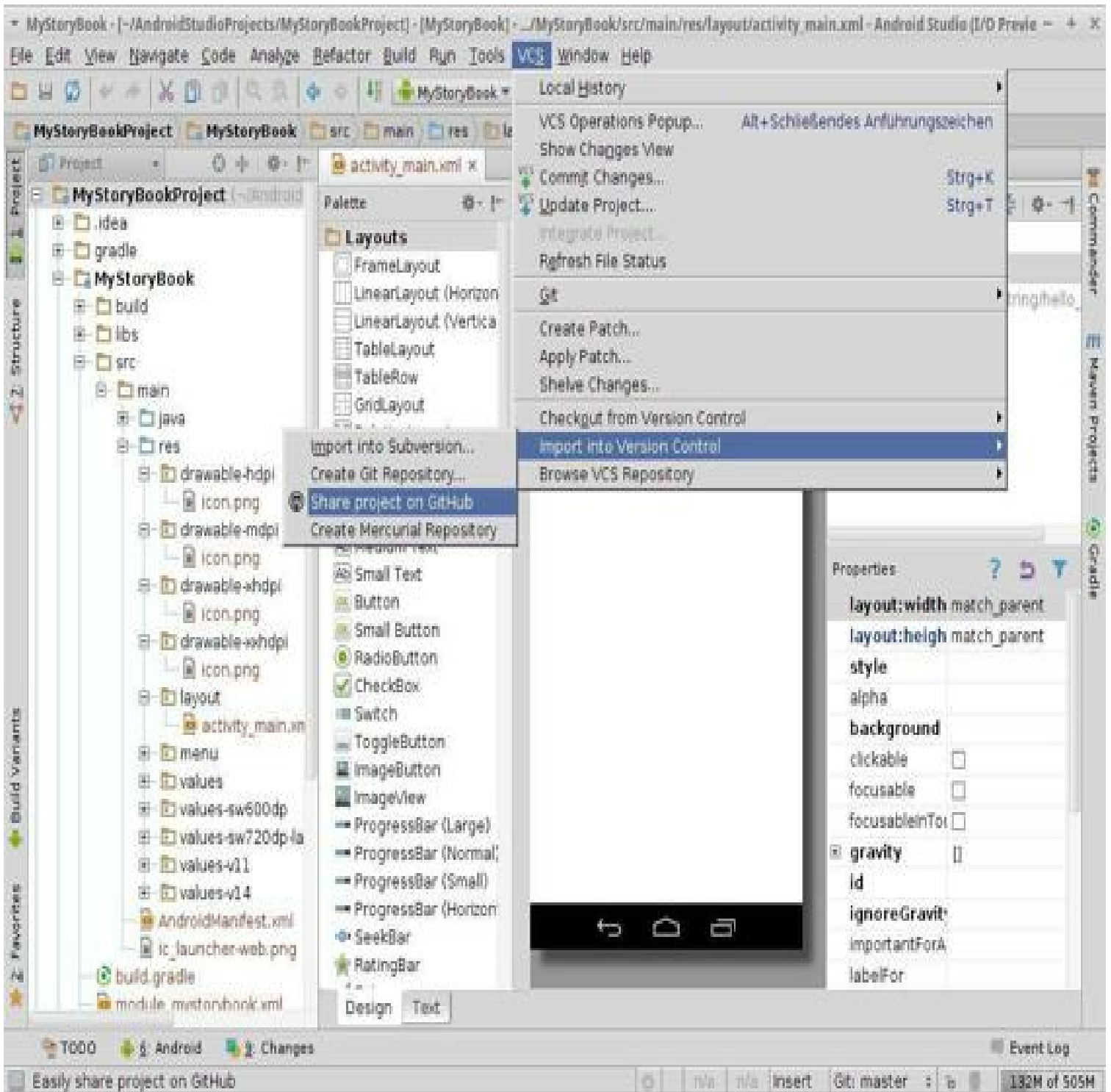


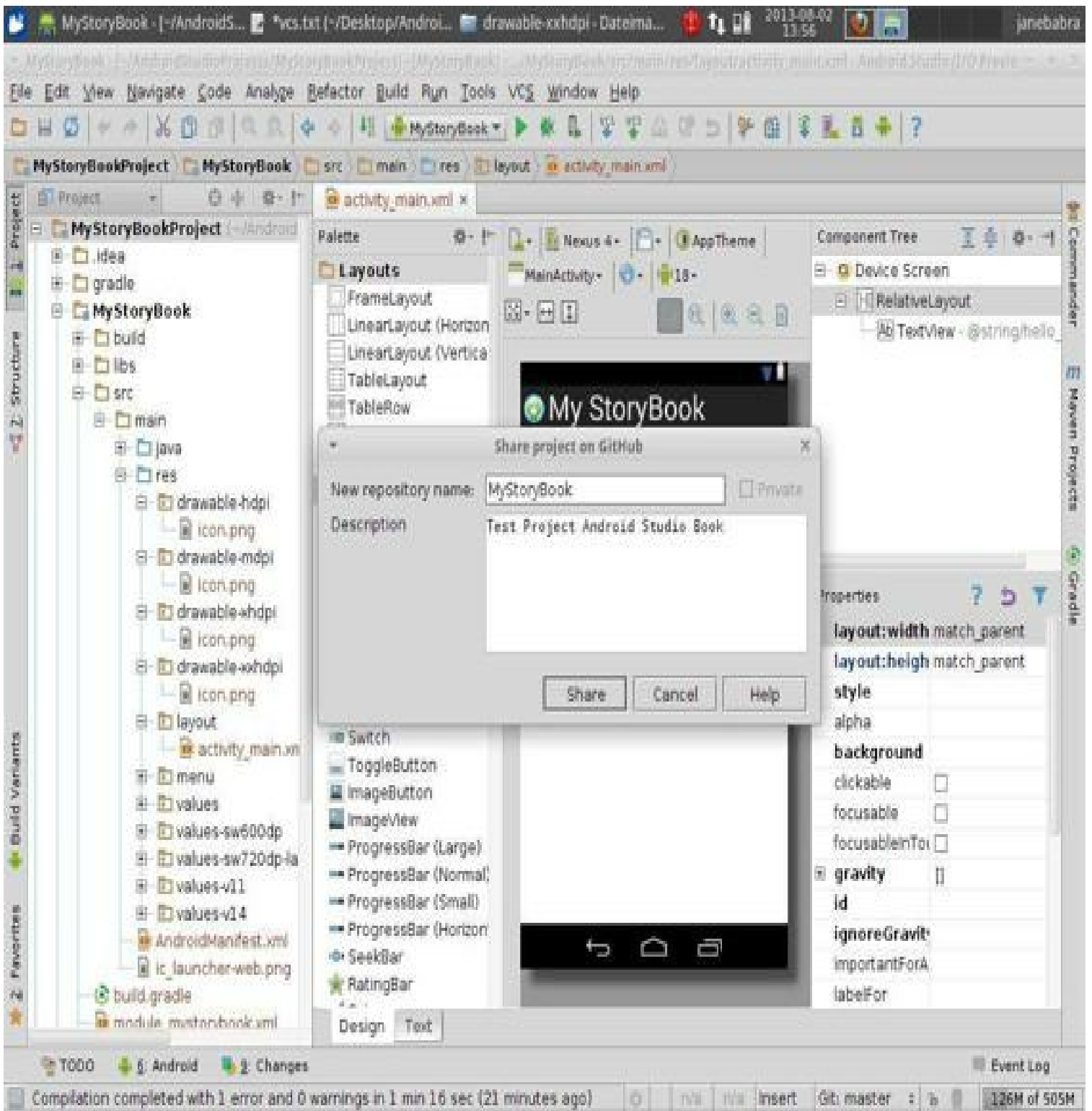
To activate VCS for your project go to the Menu VCS and select "Enable Project for Version Control Integration".





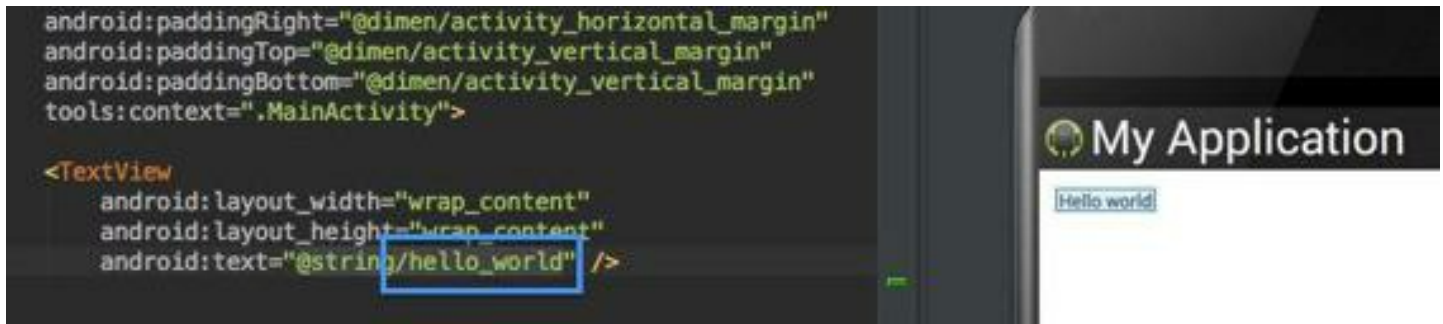
After connecting the project to GitHub, the Menu VCS has got more functionality like "Share project on GitHub".





App Layout Design: Text / Design Editor

Android Center offers an advanced layout editor, which allows you to drag-and-drop widgets in your layout and preview your layout while editing the XML.

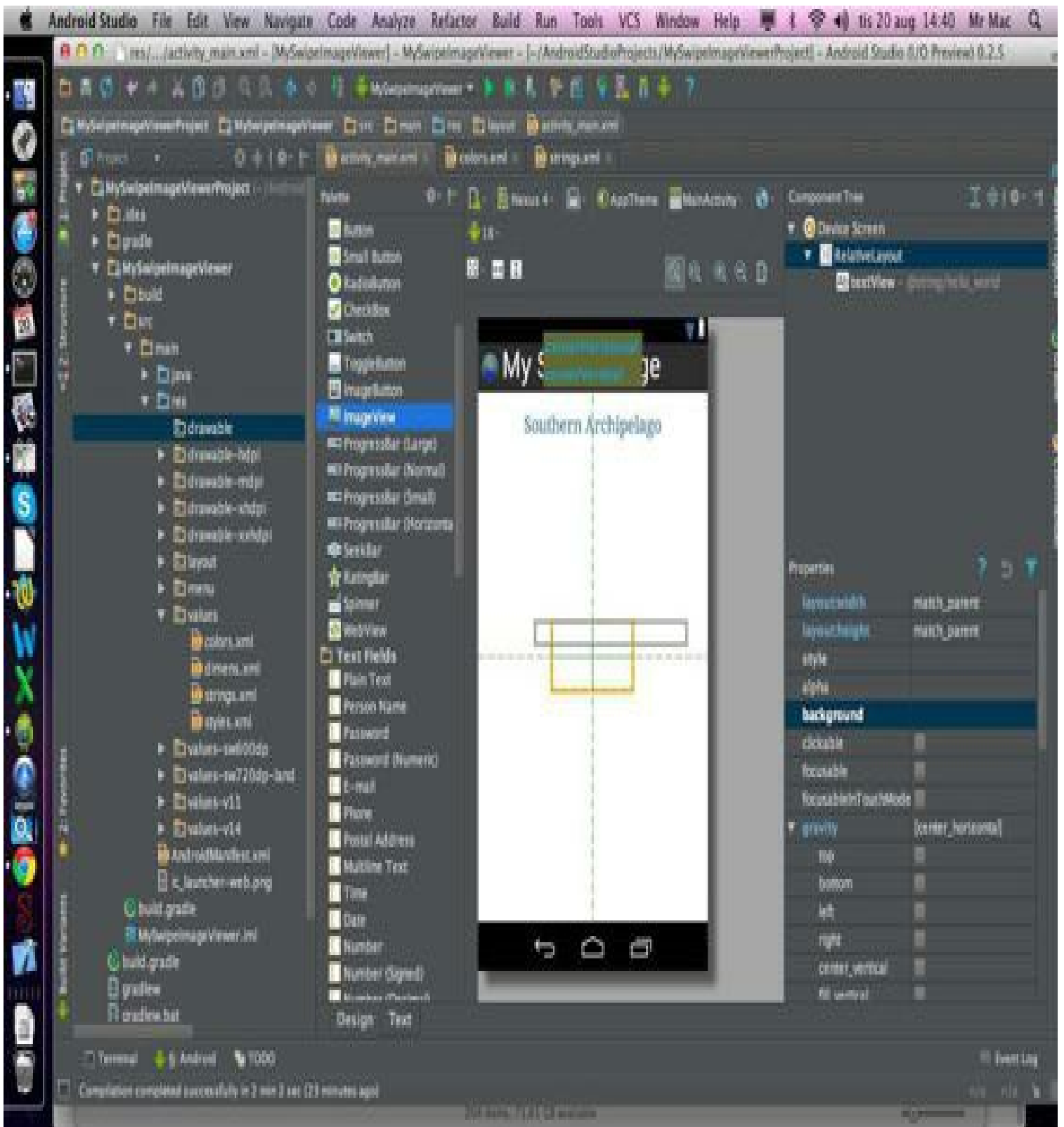


While editing in the Text View, you can preview the layout on devices by opening the preview window on the right side of the window. In the preview window, you can preview by changing various options at the top of the range, including the preview device, layout, theme, platform version, and more. To view the layout on multiple devices simultaneously, select "Preview all screen sizes" from the drop-down menu icon.

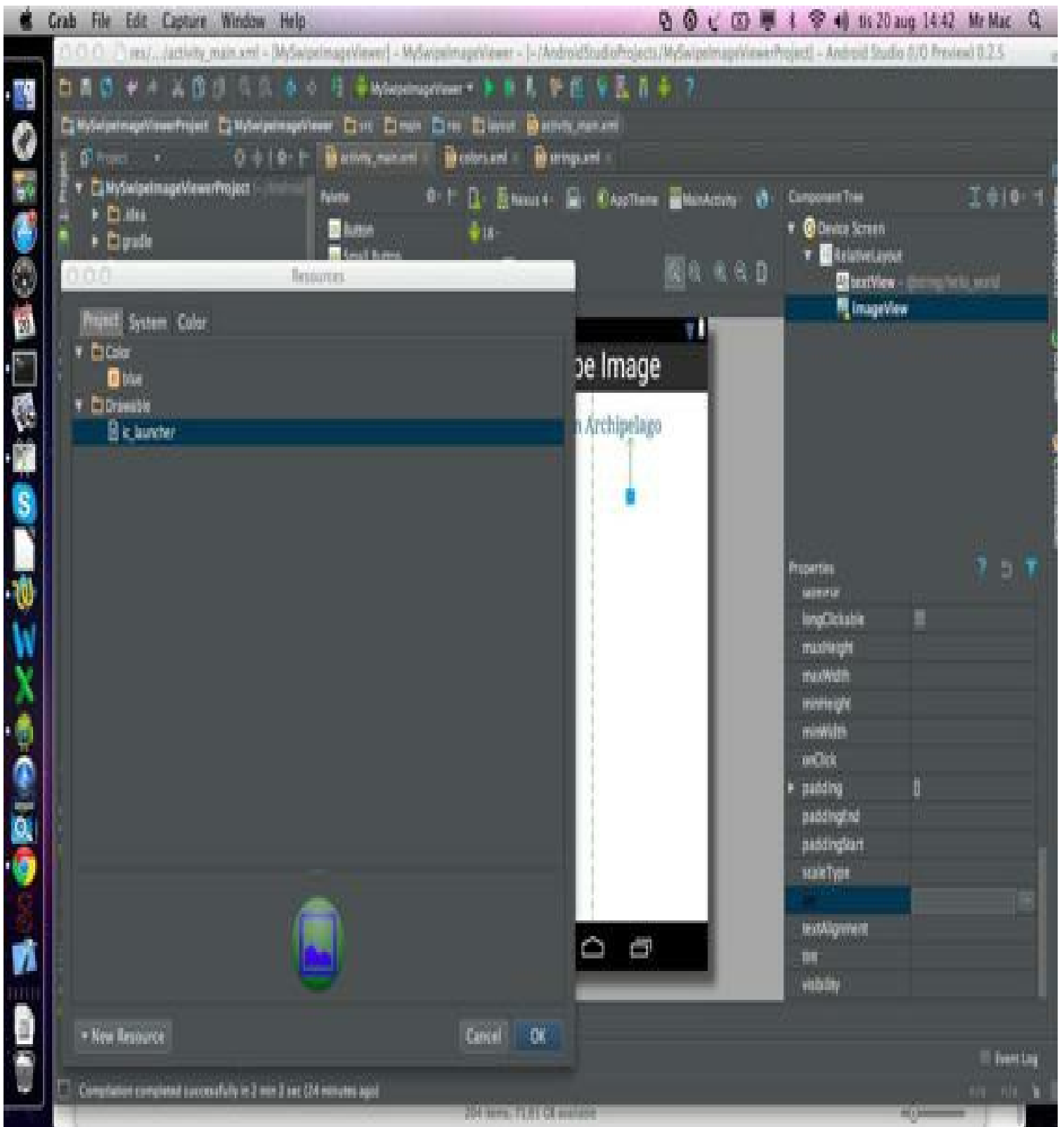
You can switch to the graphical editor by clicking on the design button at the bottom of the window. While editing in the Design view, you can show or hide the available widgets using drag-and-drop widgets by clicking on the left side of the window. Click Designer on the right side of the window to see the layout hierarchy and a list of properties for each view in the layout.

Example Android App Project: MySwipeImageViewer

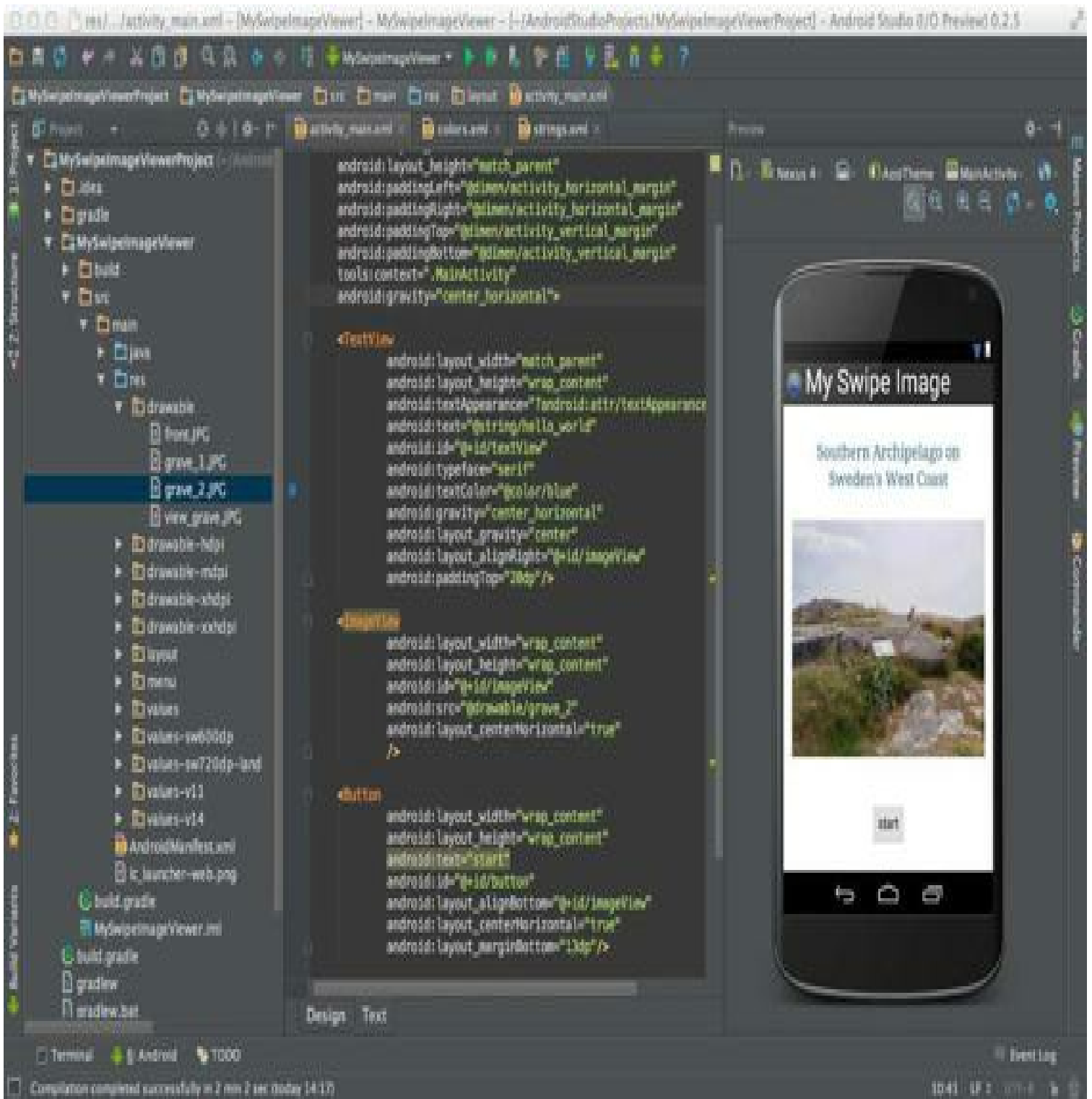
After changing the text for this app, a image is being added into the middle of the view using the design modus of the editor.



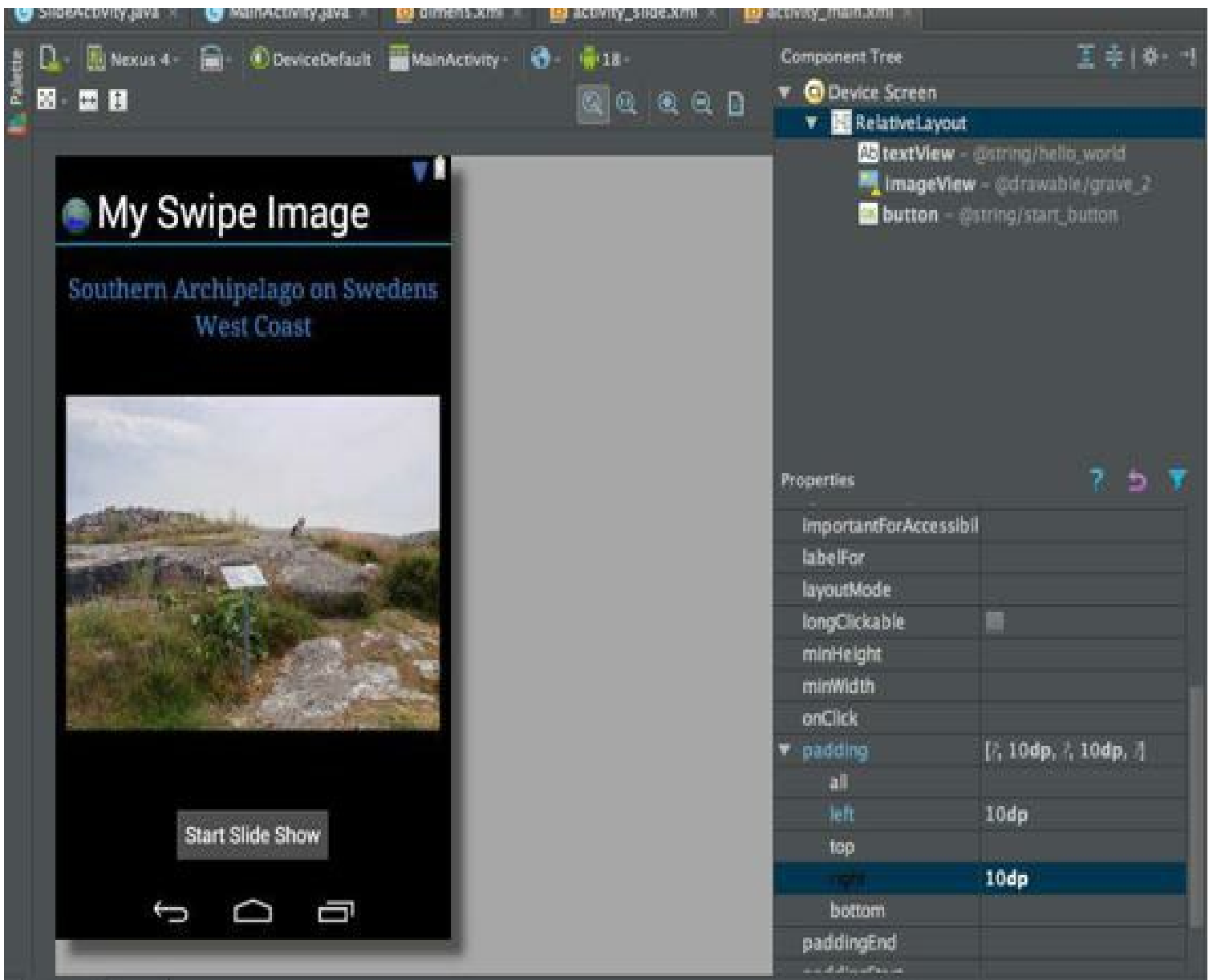
You can only select image resources that have been previously copied to the appropriate folder.



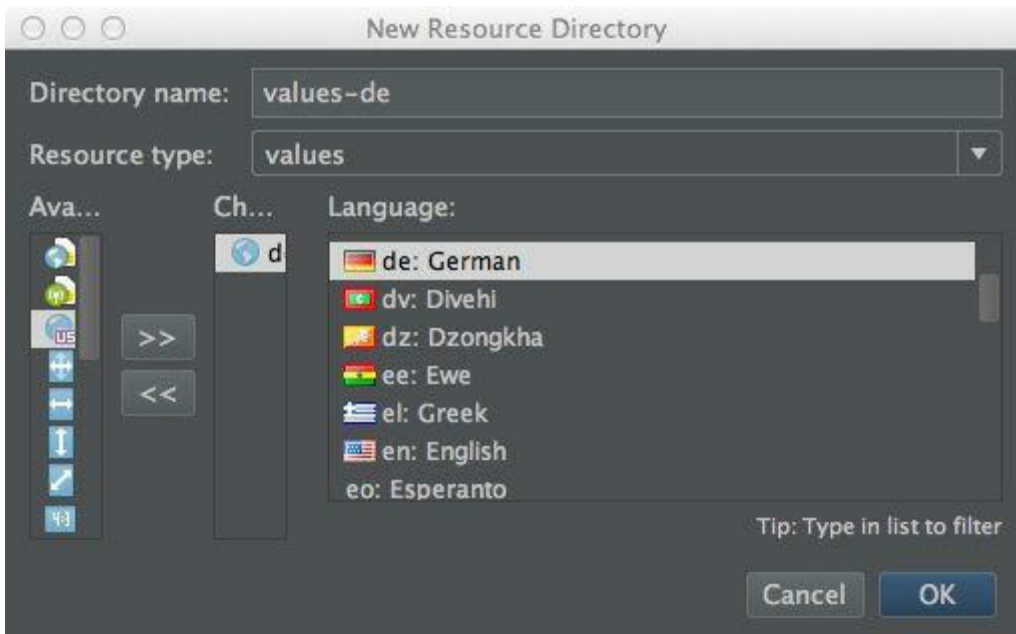
You can switch anytime between text and design mode. If you are in text mode, a live preview of the layout is automatically displayed.



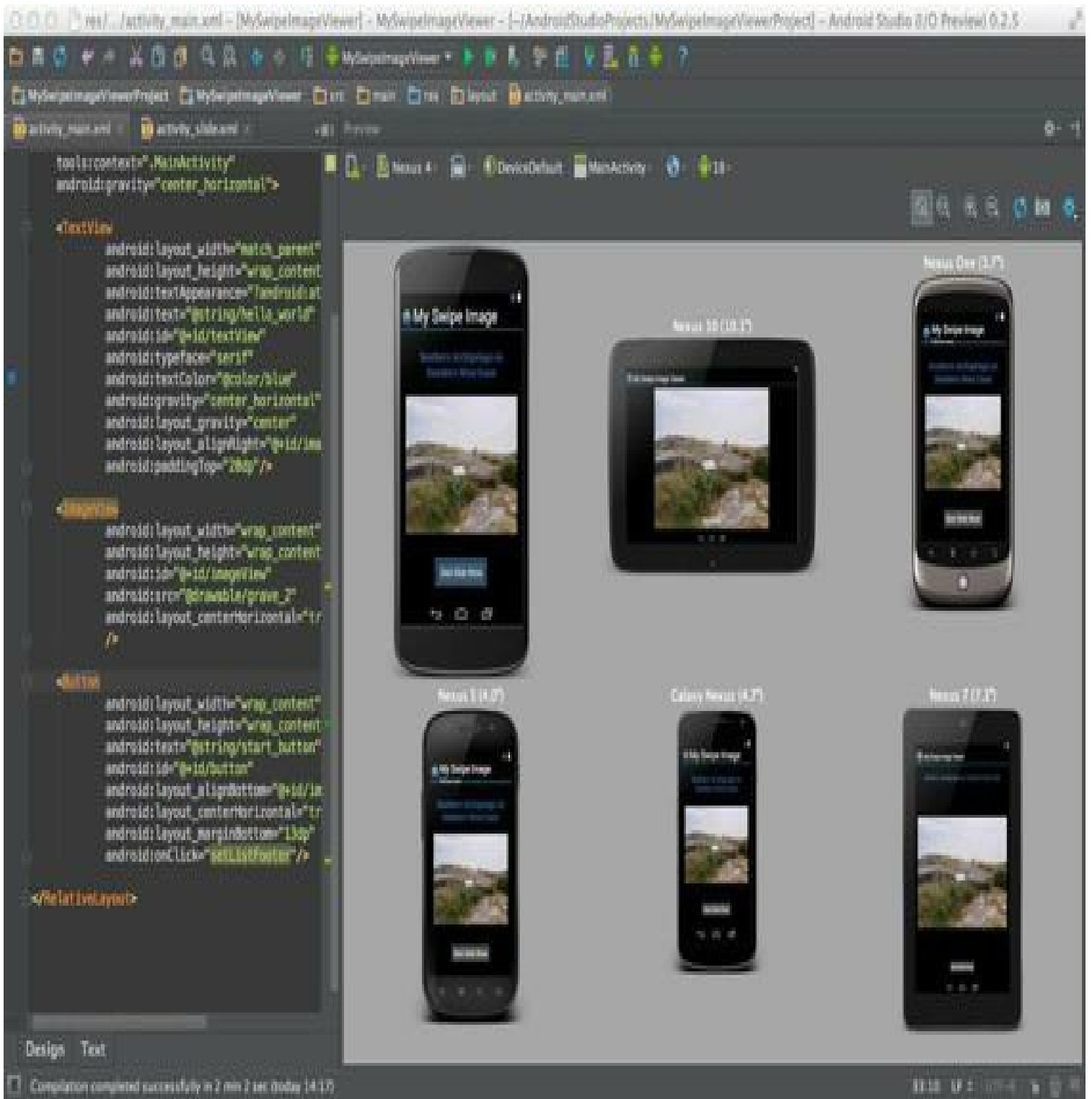
Resources such as text for the buttons or dimensions can be created on the fly, as already described in chapter 6.



To create a value folder for another language, right click on the res/ folder and select "Android resource directory".



If you want to see how the layout looks on different screen sizes, you can also do this directly from the editor by clicking in the Preview window, click the icon in the upper left corner and select for example "Preview All Screen Sizes".

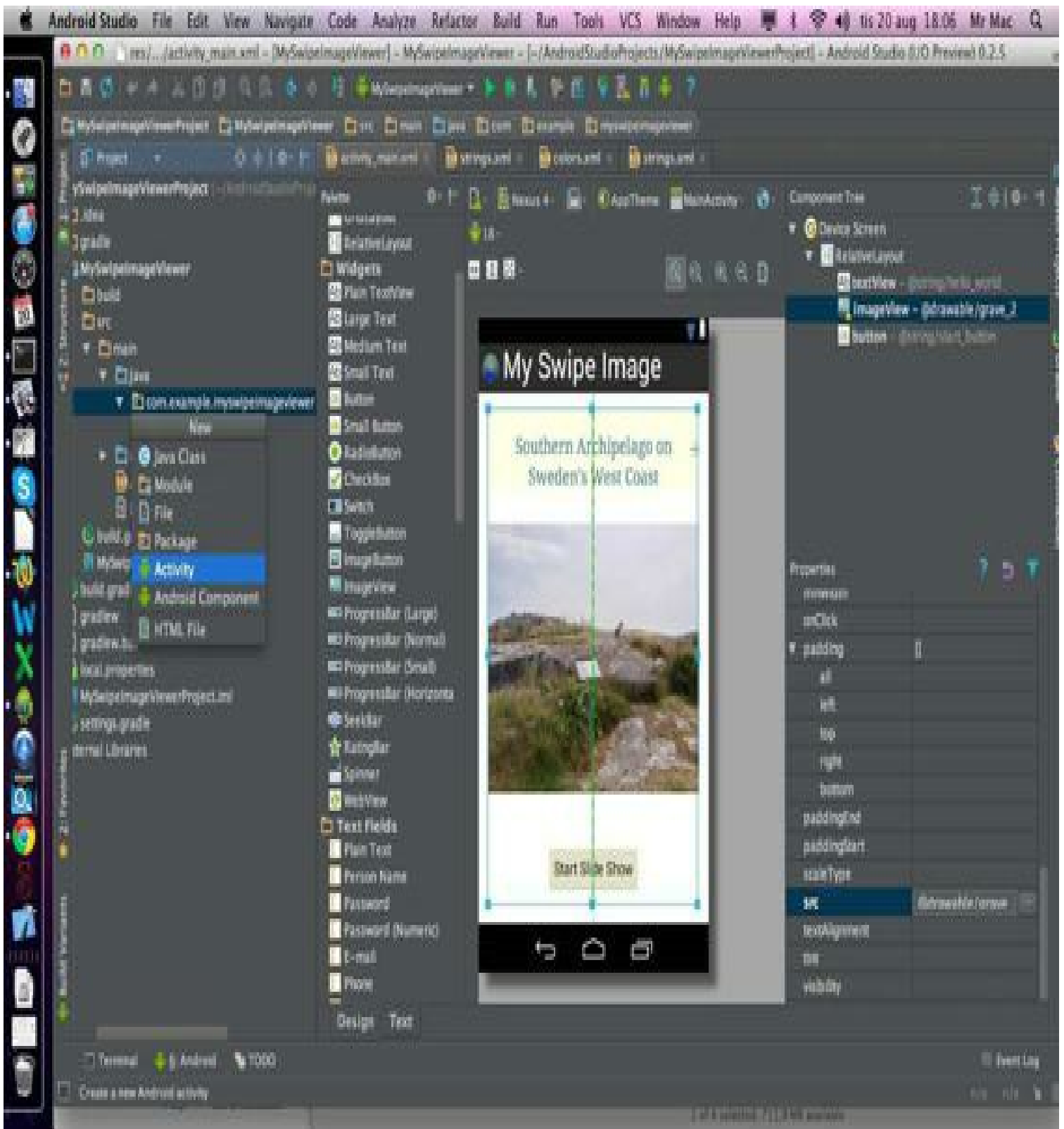


Making the App interactive (Activities)

It continues with the project app: MySwipeImageViewer.

This app has 2 Activities, the Start screen with the button „Start Slide Show“ on it, which brings us to the next Activity "the Slide Show".

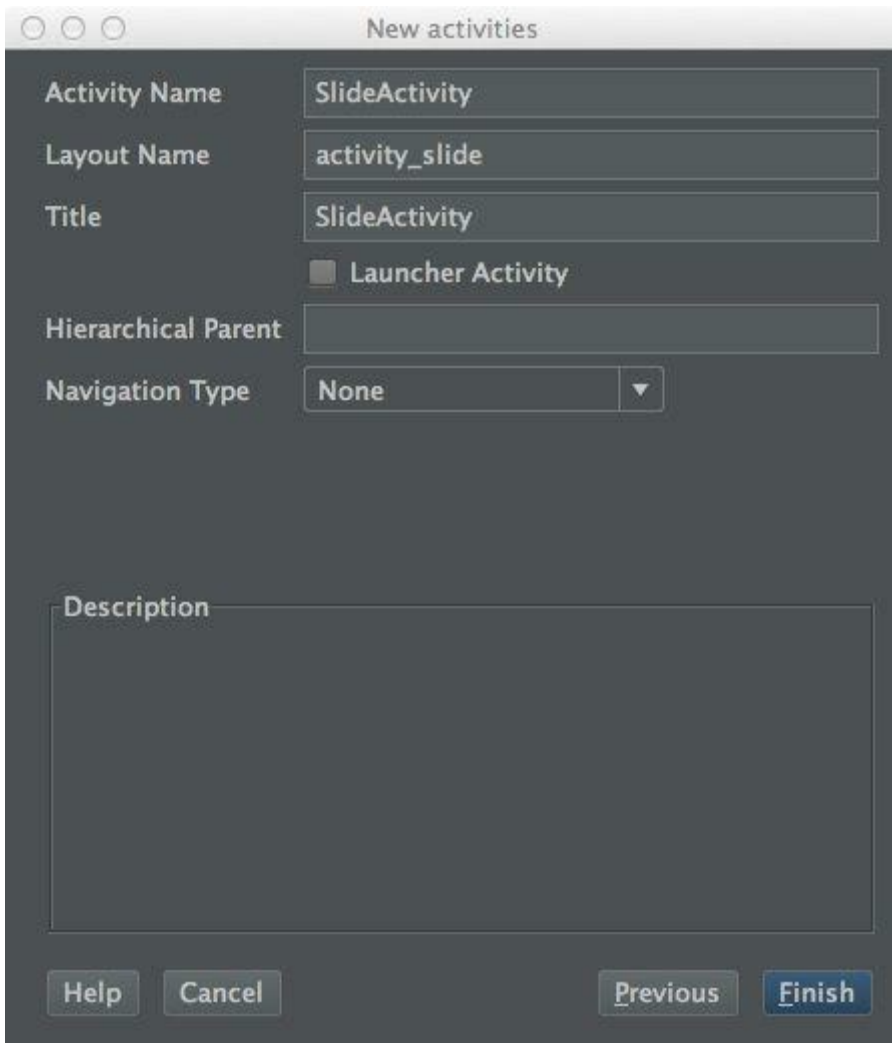
To create a new Activity right click on the package name -> New -> Java Class.



Select the standard Activity "Blank Activity".



Choose a name for the Activity and the layout file.

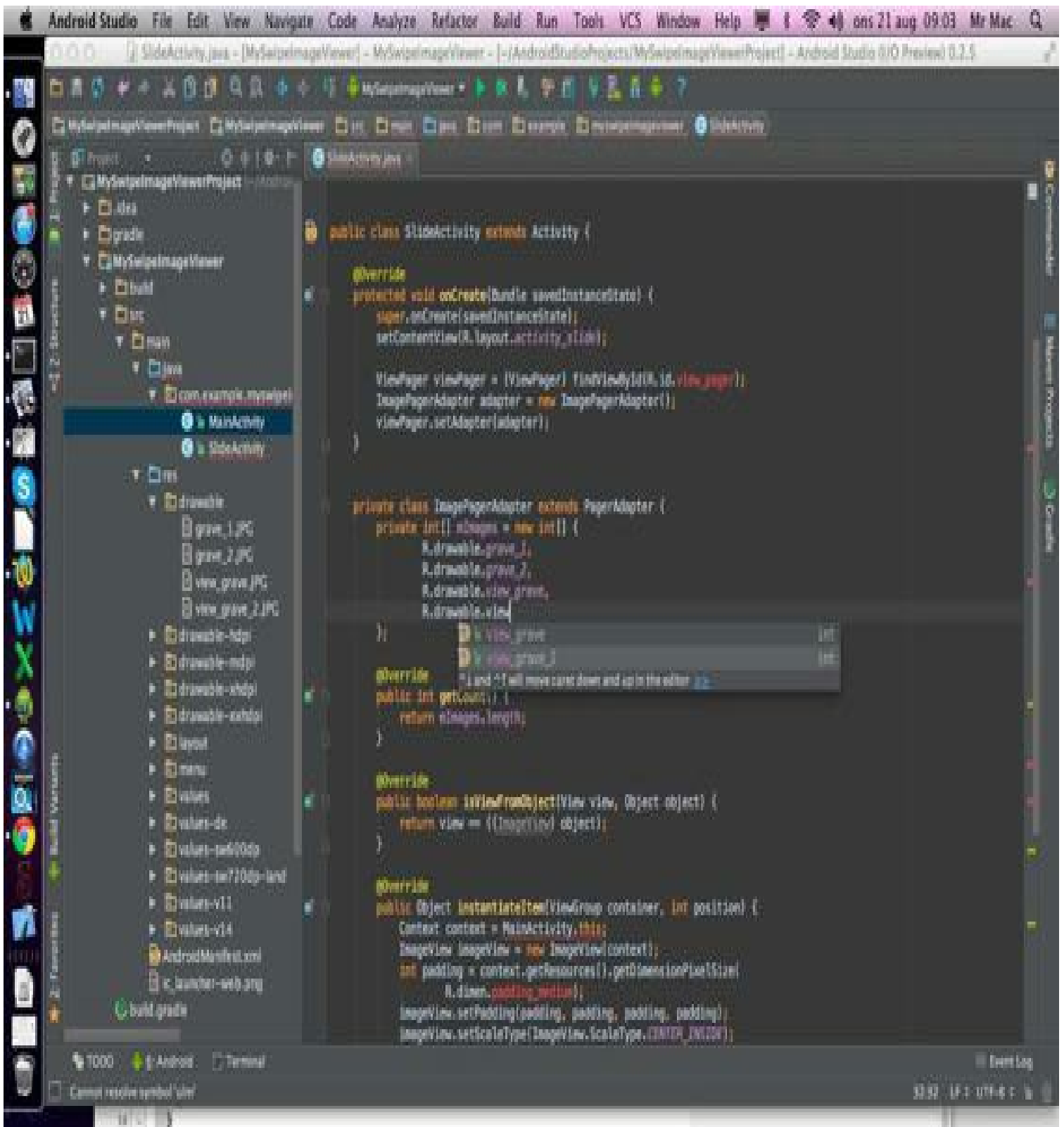


Add some new imports to the new Activity.

```
import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;

import android.content.Context;
import android.support.v4.view.PagerAdapter;
import android.support.v4.view.ViewPager;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ImageView;
```

Now we add some images to SlideActivity code for the Image Slide Show.



In the MainActivity the button is added, that connect the MainActivity to the SlideActivity.

```

package com.example.myswipeimageviewer;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
import android.content.Intent;
import android.widget.Button;
import android.view.View;
import android.view.View.OnClickListener;

public class MainActivity extends Activity {

    // The variable for the Button
    Button buttonStart;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // Get the view from activity_main.xml
        setContentView(R.layout.activity_main);

        // Locate the button in activity_main.xml
        buttonStart=(Button)findViewById(R.id.button);

        // Capture button clicks
        buttonStart.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                // Go to the new Activity to see the Dia Show
                Intent intent = new Intent(MainActivity.this, SlideActivity.class);
                startActivity(intent);
            }
        });
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.main, menu);
    }
}

```

In the AndroidManifest.xml has already both Activities. Nothing to do here.

```

<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.mySwipeImageviewer"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="15"
        android:targetSdkVersion="16" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="My Swipe Image Viewer"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.example.mySwipeImageviewer.MainActivity"
            android:label="My Swipe Image Viewer" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity
            android:name="com.example.mySwipeImageviewer.SlideActivity"
            android:label="SlideActivity" >
        </activity>
    </application>
</manifest>

```

Der Source-Code for this Project can be downloaded from GitHub:

<https://github.com/janebabra/MySwipeImageViewerProject>

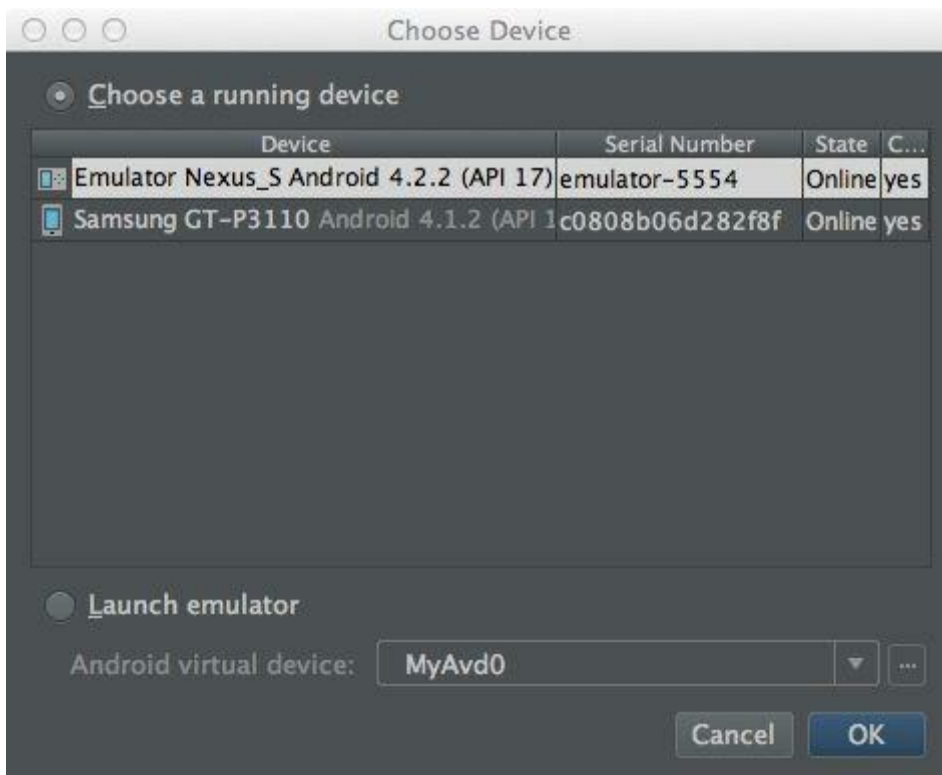
Build and Run the App

Run the app is just a matter of clicking on the green triangle icon on the menu. Android Studio is first compile your code and then makes it available for execution.

The progress can be observed in the status bar at the bottom of the screen.

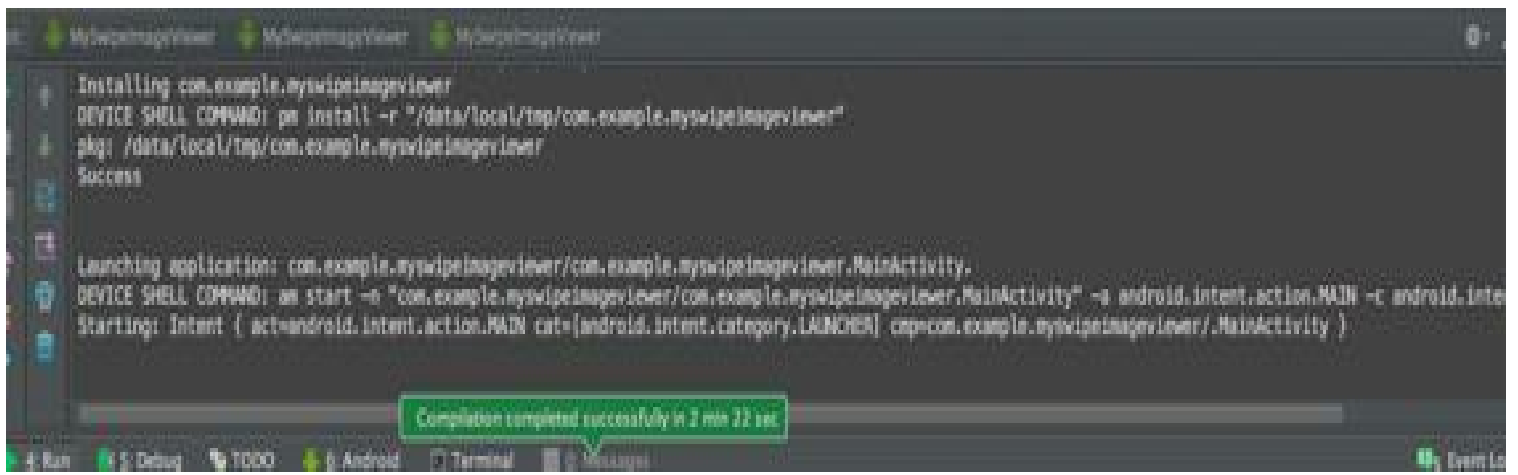
If there are error messages caused by problems in the code opens a log window at the bottom of the screen.

If the code is ready to run on a device, opens the window "choose device". Here you see in the top list, emulators and connected android devices that are already running.



If you don't want to use any of the current devices, go to the lower half of the window, click "Launch Emulator" and choose a matching from the "drop-down menu. ""

The app will run finally on the selected emulator or Android device.



Running the app from the command line

The fact that the Android studio has the Gradle build system integrated, gives more opportunities to work from the command line.

To create an unsigned app as a Debug and a Release APK proceed as follows:

Go in the terminal window, cd to the directory MySwipeImageViewerProject.

For Windows:

```
gradlew.bat assemble
```

For Linux und Mac:

```
./gradlew assemble
```

After that, the two files are build and stored under:

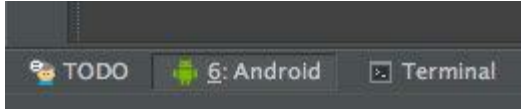
```
/MySwipeImageViewerProject/ MySwipeImageViewer/build/apk
```

```
MySwipeImageViewer-debug-unaligned.apk and
```

```
MySwipeImageViewer-release-unsigned.apk
```

Debugging

When creating and executing an application with Android Studio, you can see the adb and device log messages (logcat) in the DDMS view pane by clicking on Android at the bottom of the window.



If you want to debug your app with the Android debug monitor, you can do this by click the symbol



from the toolbar.

In the debug monitor, you can see the complete set of tools for DDMS and find out more about the behavior of your app. It also includes the Hierarchy Viewer tools to optimize your layout.

These tools are the same as in the ADT for Eclipse.

There's one exception. The Traceview from the ADT has been replaced in Android Studio 0.2.10 with his own viewer.

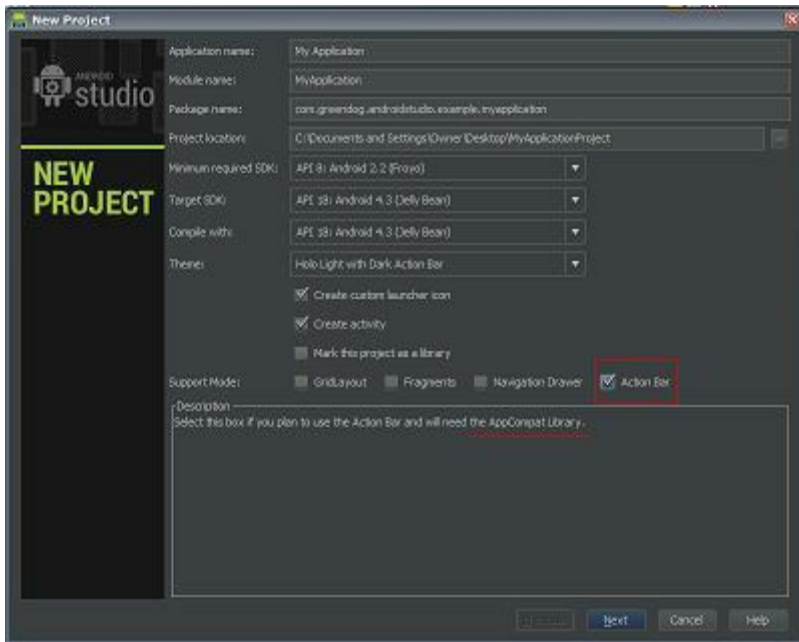
In the Android DDMS select the Start Method Tracing symbol to launch this viewer.

Testing

The following is a test project and the tests are carried out both by command line level as well as by the IDE.

As a basis to create a project like the "MyApplicationProject" from the chapter "Creating a new Android app project" has been used.

The name of the test project is *MyApplicationTestProject*.



As seen in the screenshots, a "Hello World" template is selected with Action Bar, changed the text and added a background image. So, the following picture shows the start position.



Of the just created project open the *activity_main.xml* in the Editor to check whether the *TextView* includes an "id", as will be later referred to it in the test. If no "id" exists, as shown in this screenshot, create one.



The file can then be closed again.

The file structure for the test has to be created manually. To do this, go in the Project View panel.

In the project folder, right-click on *src /* and select from the context menu

New -> Select Directory.

Name the folder *instrumentTest* and confirm with <OK>.

Right-click now on the *instrumentTest* folder to create a new folder:

New -> Select Directory and name the subdirectory *java*.

Now click with the right mouse button on the *java* folder and with

New -> Select Package create a package for the Test Activity.

The name of this package should be the same like the main package of the project with an attached **.test**.

In this example, the package is:

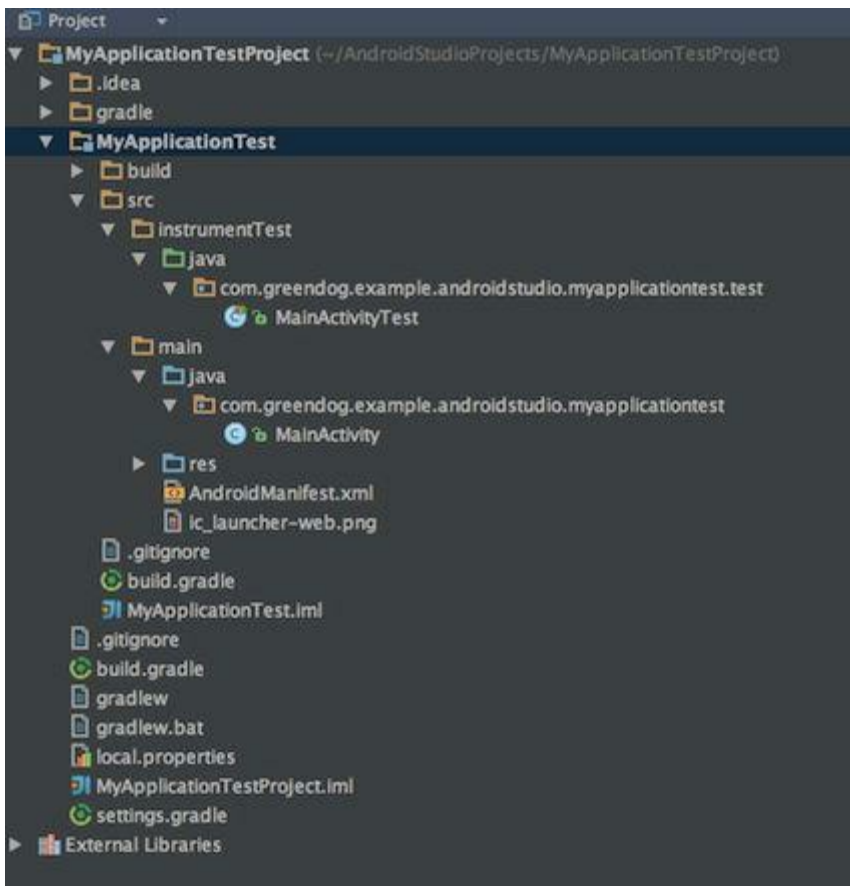
com.greendog.example.androidstudio.myapplicationtest.test

Add to the just created package a test activity. Right-click on the package name and select

New -> Java class

Name the new Activity *MainActivityTest*.

The project structure should now look like this:



The here used test activity has the task to check, whether the TextView appears with the text contained on the screen.

```
1 package com.greendog.example.androidstudio.myapplicationtest.test;
2
3 import android.annotation.TargetApi;
4 import android.os.Build;
5 import android.test.ActivityInstrumentationTestCase2;
6 import android.widget.TextView;
7 import static android.test.ViewAsserts.assertOnScreen;
8
9 import com.greendog.example.androidstudio.myapplicationtest.MainActivity;
10 import com.greendog.example.androidstudio.myapplicationtest.R;
11
12 public class MainActivityTest extends ActivityInstrumentationTestCase2<MainActivity> {
13
14     private MainActivity mMainActivity;
15     private TextView mHelloTestWorldTextView;
16
17     @TargetApi(Build.VERSION_CODES.FROYO)
18     public MainActivityTest() {
19         super(MainActivity.class);
20     }
21 }
22
23 protected void setUp() throws Exception {
24     super.setUp();
25
26     mMainActivity = getActivity();
27     mHelloTestWorldTextView = (TextView) mMainActivity.findViewById(R.id.textView2);
28 }
29
30 public void testTextView() {
31     assertOnScreen(mMainActivity.getWindow().getDecorView(), mHelloTestWorldTextView);
32 }
33 }
34
35 }
```

The project is now ready for the scheduled test.

Initially, the project must run on a device or emulator via an USB-connected phone / tablet. Then you can either from the IDE or from the command line start and run the tests.

For the test from the command line navigate to the project directory, like you would to build the project.

The test command is:

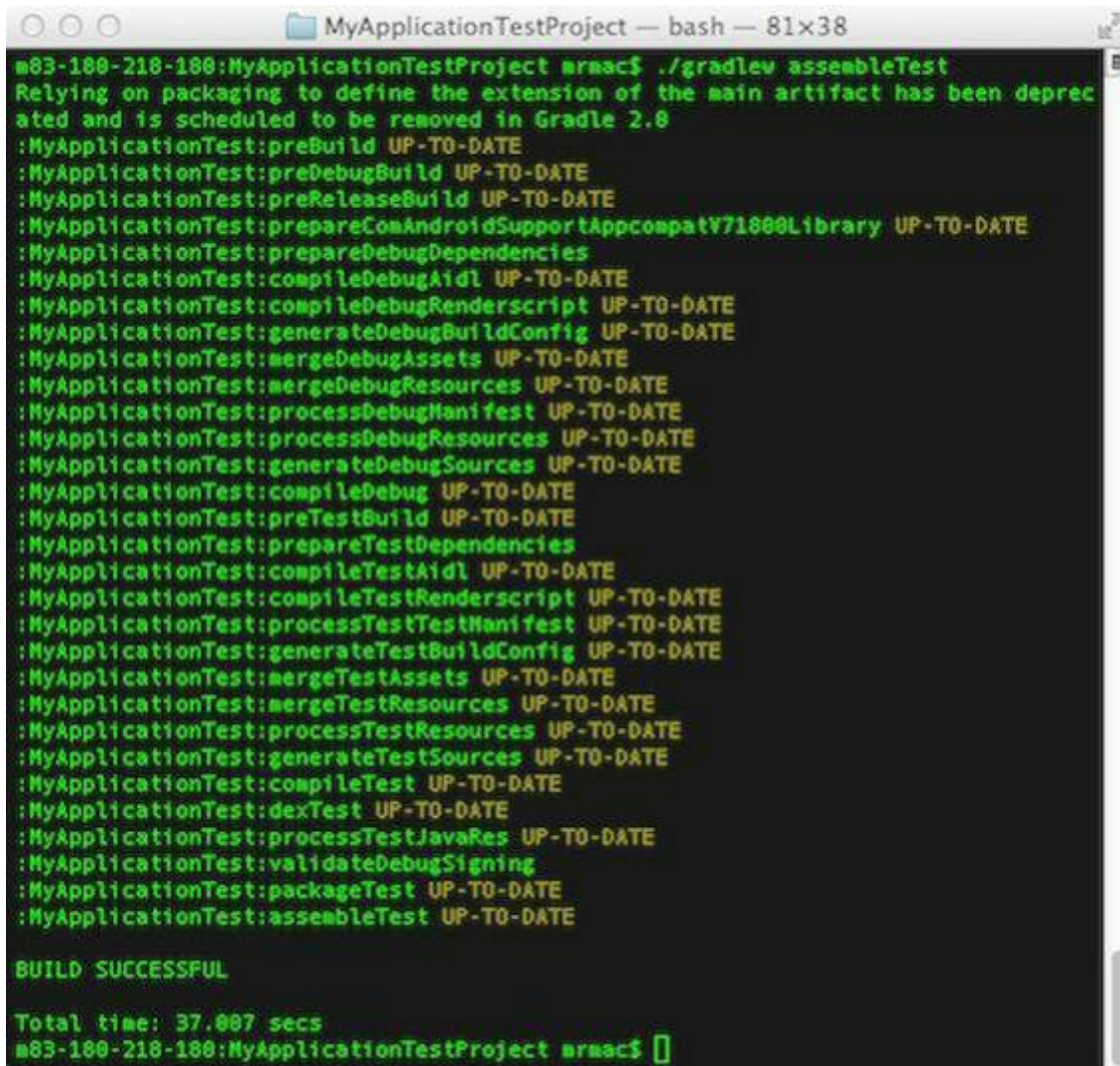
For Linux and Mac OS X:

```
./gradlew assembleTest
```

For Windows:

```
gradlew.bat assembleTest
```

A successful test run would be look like this:



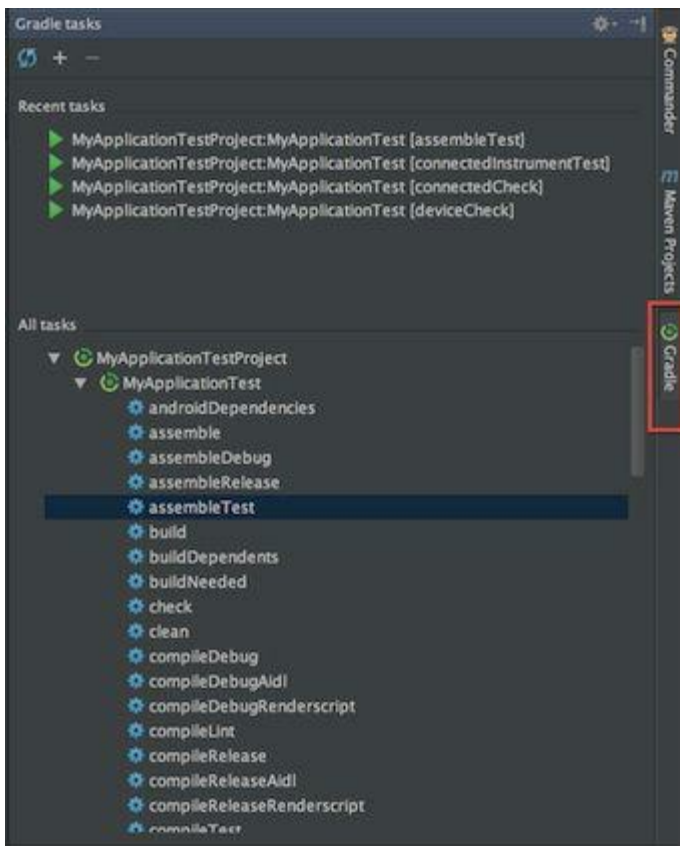
```
MyApplicationTestProject — bash — 81x38
m83-180-218-180:MyApplicationTestProject armac$ ./gradlew assembleTest
Relying on packaging to define the extension of the main artifact has been deprecated and is scheduled to be removed in Gradle 2.8
:MyApplicationTest:preBuild UP-TO-DATE
:MyApplicationTest:preDebugBuild UP-TO-DATE
:MyApplicationTest:preReleaseBuild UP-TO-DATE
:MyApplicationTest:prepareComAndroidSupportAppcompatV71880Library UP-TO-DATE
:MyApplicationTest:prepareDebugDependencies
:MyApplicationTest:compileDebugAidl UP-TO-DATE
:MyApplicationTest:compileDebugRenderscript UP-TO-DATE
:MyApplicationTest:generateDebugBuildConfig UP-TO-DATE
:MyApplicationTest:mergeDebugAssets UP-TO-DATE
:MyApplicationTest:mergeDebugResources UP-TO-DATE
:MyApplicationTest:processDebugManifest UP-TO-DATE
:MyApplicationTest:processDebugResources UP-TO-DATE
:MyApplicationTest:generateDebugSources UP-TO-DATE
:MyApplicationTest:compileDebug UP-TO-DATE
:MyApplicationTest:preTestBuild UP-TO-DATE
:MyApplicationTest:prepareTestDependencies
:MyApplicationTest:compileTestAidl UP-TO-DATE
:MyApplicationTest:compileTestRenderscript UP-TO-DATE
:MyApplicationTest:processTestTestManifest UP-TO-DATE
:MyApplicationTest:generateTestBuildConfig UP-TO-DATE
:MyApplicationTest:mergeTestAssets UP-TO-DATE
:MyApplicationTest:mergeTestResources UP-TO-DATE
:MyApplicationTest:processTestResources UP-TO-DATE
:MyApplicationTest:generateTestSources UP-TO-DATE
:MyApplicationTest:compileTest UP-TO-DATE
:MyApplicationTest:dexTest UP-TO-DATE
:MyApplicationTest:processTestJavaRes UP-TO-DATE
:MyApplicationTest:validateDebugSigning
:MyApplicationTest:packageTest UP-TO-DATE
:MyApplicationTest:assembleTest UP-TO-DATE

BUILD SUCCESSFUL

Total time: 37.007 secs
m83-180-218-180:MyApplicationTestProject armac$
```

To access test from the IDE, just click on the right side of the window on <Gradle> to see the available Gradle tasks.

Select a command from the list and double-click to start the desired test.



The test results are stored under the *build/reports/* in HTML format and can be viewed in the web browser.



A click on the package name takes us to the page that shows specifically what was tested.

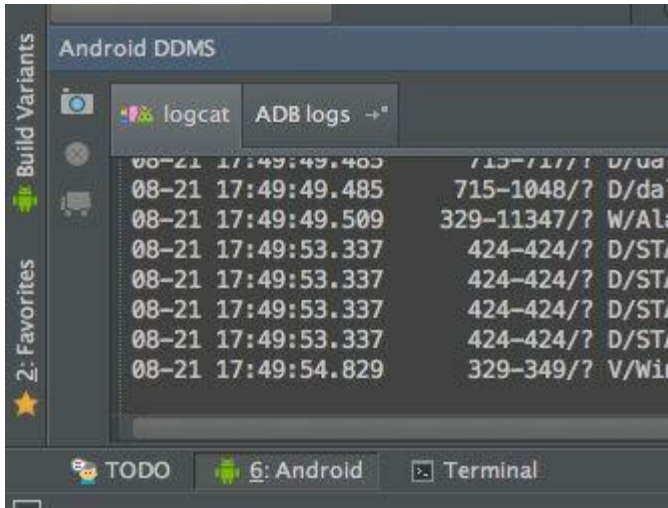


The project is on my GitHub, see Links.

Preparing the app for the Android Market

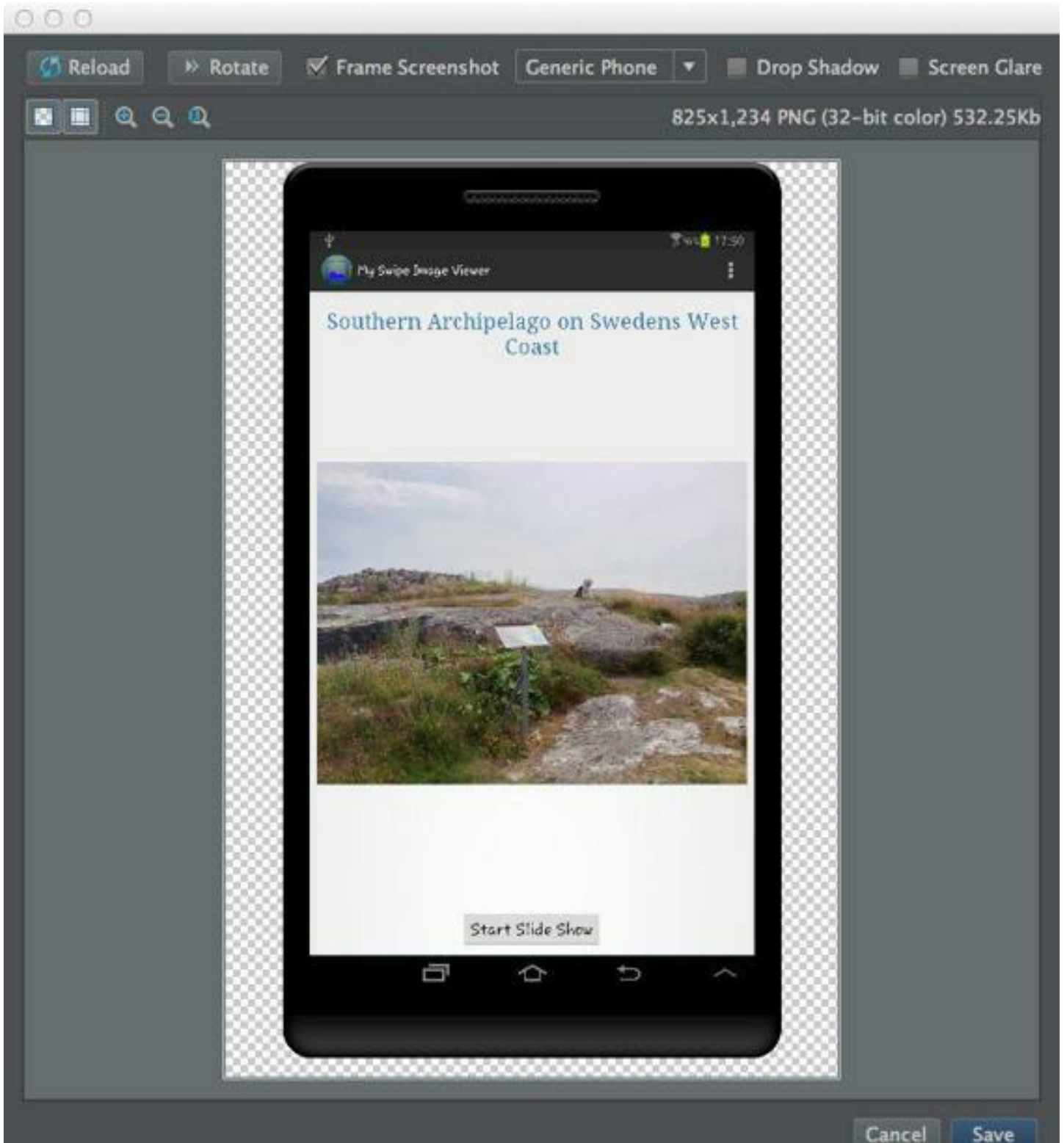
Screenshots

The easiest way to do this is from the „Android DDMS“ .



Run the app on an Emulator or on a device. Then click on the camera symbol in the „Android DDMS“.

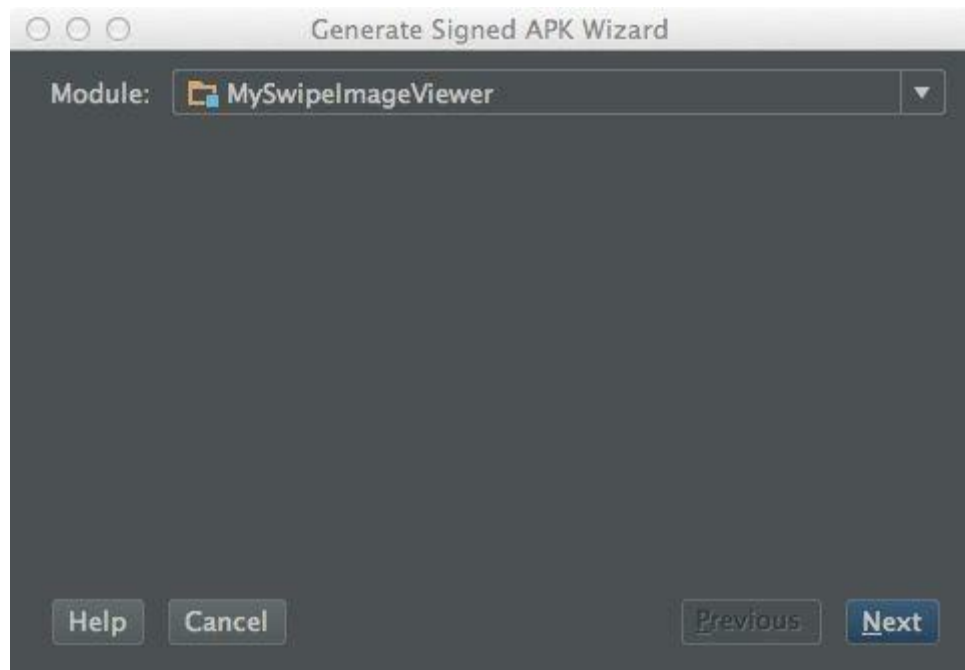
Choose from the Drop-Down menu one of the phones or tablets to get the size/ screen resolution you need.



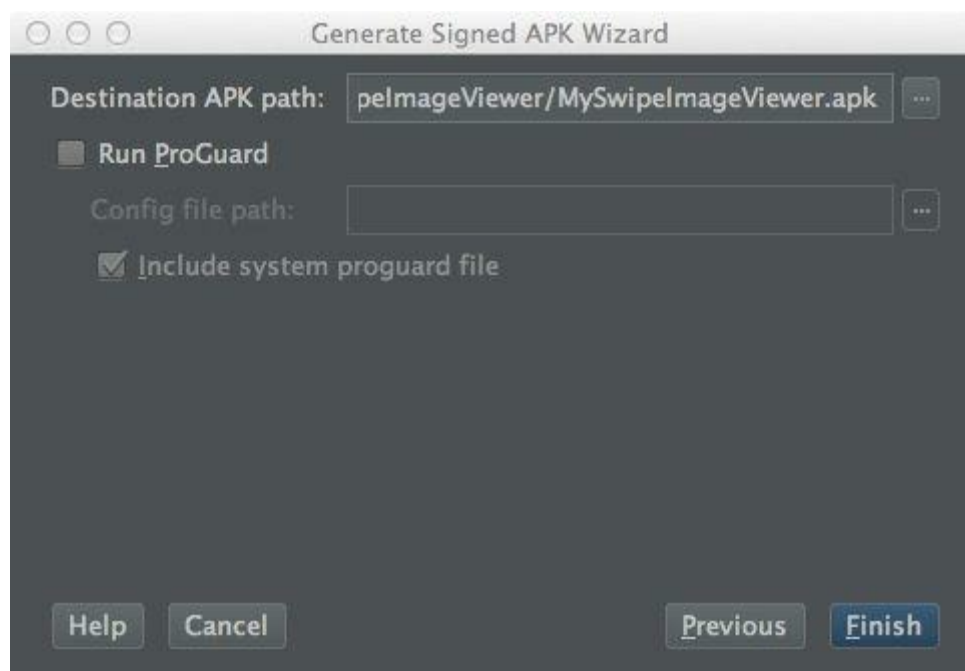
Signing the app (Wizard)

From the Build menu select “Generate Signed APK”.

This opens the window „Generate SignedAPKWizard“.



An existing key can be loaded or a new one created. Fill in all fields, there is one last step. In the case that ProGuard is to be used, the "Run ProGuard" has to be activated, than navigate to the configuration file and click <Finish>.





The app is ready to be deployed on the Android Markets.

ProGuard

For an app using ProGuard, the build.gradle file needs the following entry:

```
buildTypes
```

```
release {
```

```
runProguard true
```

```
proguardFile
```

```
getDefaultProguardFile('proguard-android.txt')
```

```
}
```

Signing a release APK with Gradle

There's nothing wrong with using the Signing Wizard, but in some cases it can be useful to automate the signing process by using Gradle. There are 3 options to do this:

First option: everything is been stored in the projects *build.gradle*

```
android {
    ...
    signingConfigs {
        release {
            storeFile file("release.keystore")
            storePassword "*****"
            keyAlias "*****"
            keyPassword "*****"
        }
    }
    buildTypes {
        release {
            signingConfig signingConfigs.release
        }
    }
}
```

The next example will prompt for each parameter:

```
signingConfigs {
    release {
        storeFile file(System.console().readLine("\n\$ Enter keystore
path: "))
        storePassword System.console().readLine("\n\$ Enter keystore
password: ")
        keyAlias System.console().readLine("\n\$ Enter key alias: ")
        keyPassword System.console().readLine("\n\$ Enter key password: ")
    }
}
```

Another solution is, to put the information into *~/.gradle/gradle.properties* like this:

```
RELEASE_STORE_FILE={path to your keystore}
RELEASE_STORE_PASSWORD=*****
```



```
RELEASE_KEY_ALIAS=*****
```

```
RELEASE_KEY_PASSWORD=*****
```

Next in the `build.gradle` file, the configuration would be look like this:

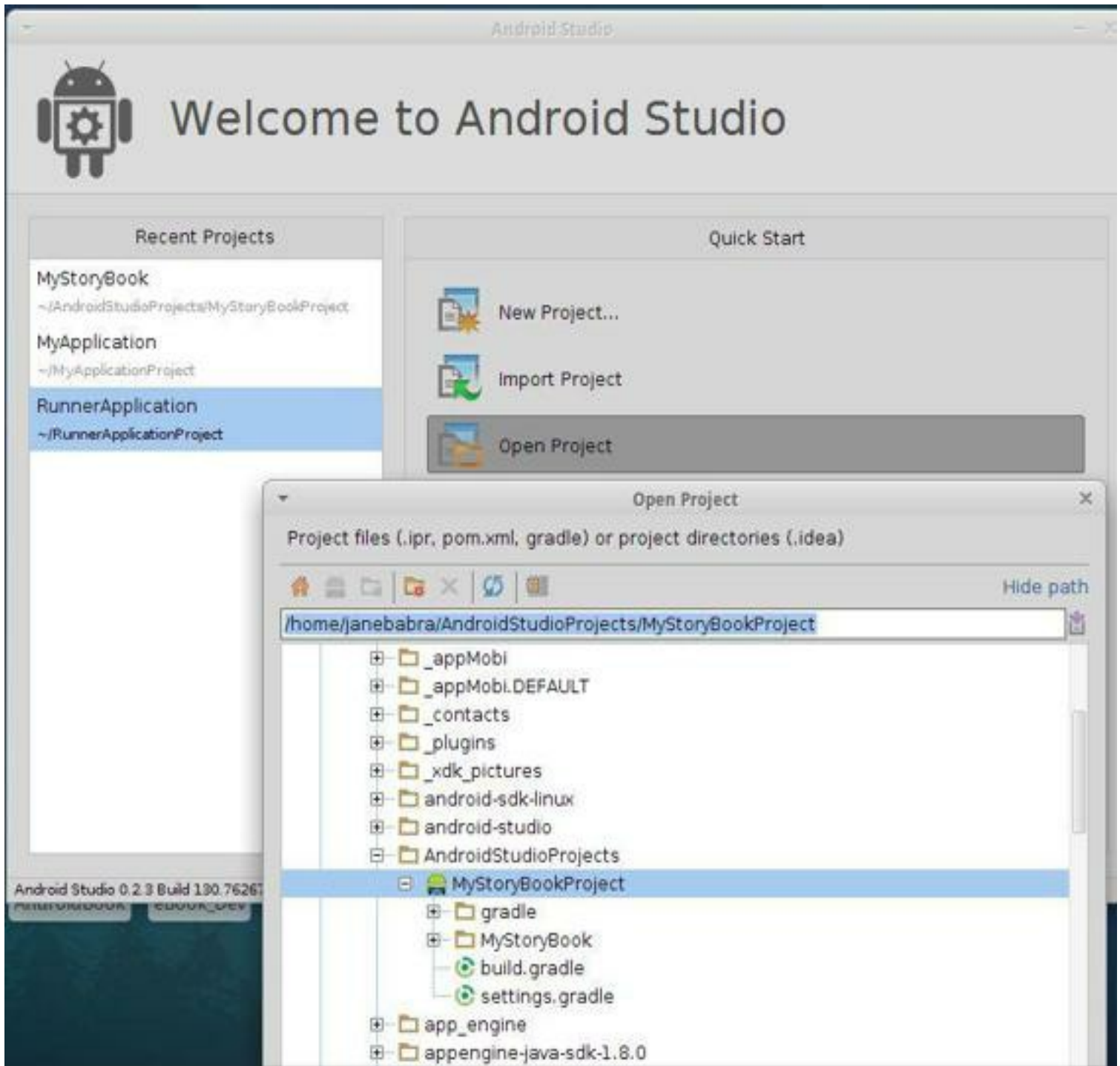
```
...
```

```
signingConfigs {  
    release {  
        storeFile file(RELEASE_STORE_FILE)  
        storePassword RELEASE_STORE_PASSWORD  
        keyAlias RELEASE_KEY_ALIAS  
        keyPassword RELEASE_KEY_PASSWORD  
    }  
}  
  
buildTypes {  
    release {  
        signingConfig signingConfigs.release  
    }  
}  
  
....
```

In all cases the command `"gradle assembleRelease"` will produce a signed release apk.

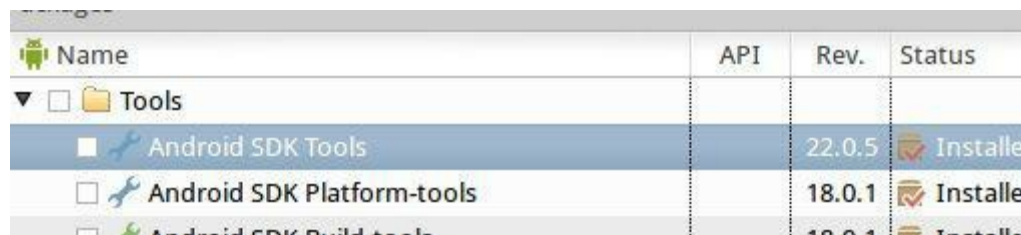
Import Android Projects

To import Android Studio projects, select from the Welcome Screen "Import Project".



Import Eclipse projects

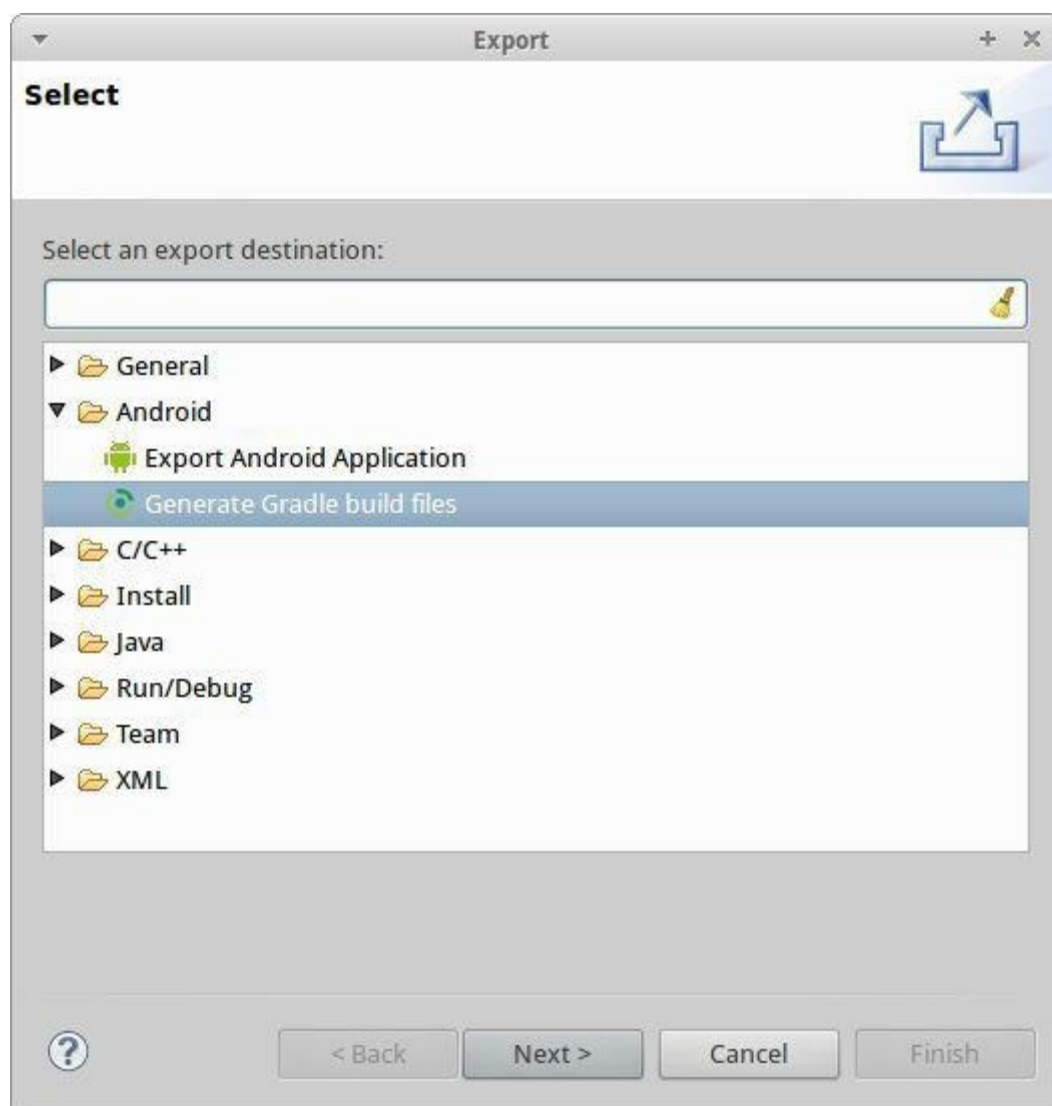
1. Eclipse must have ADT-Plugin 22.0 or higher.



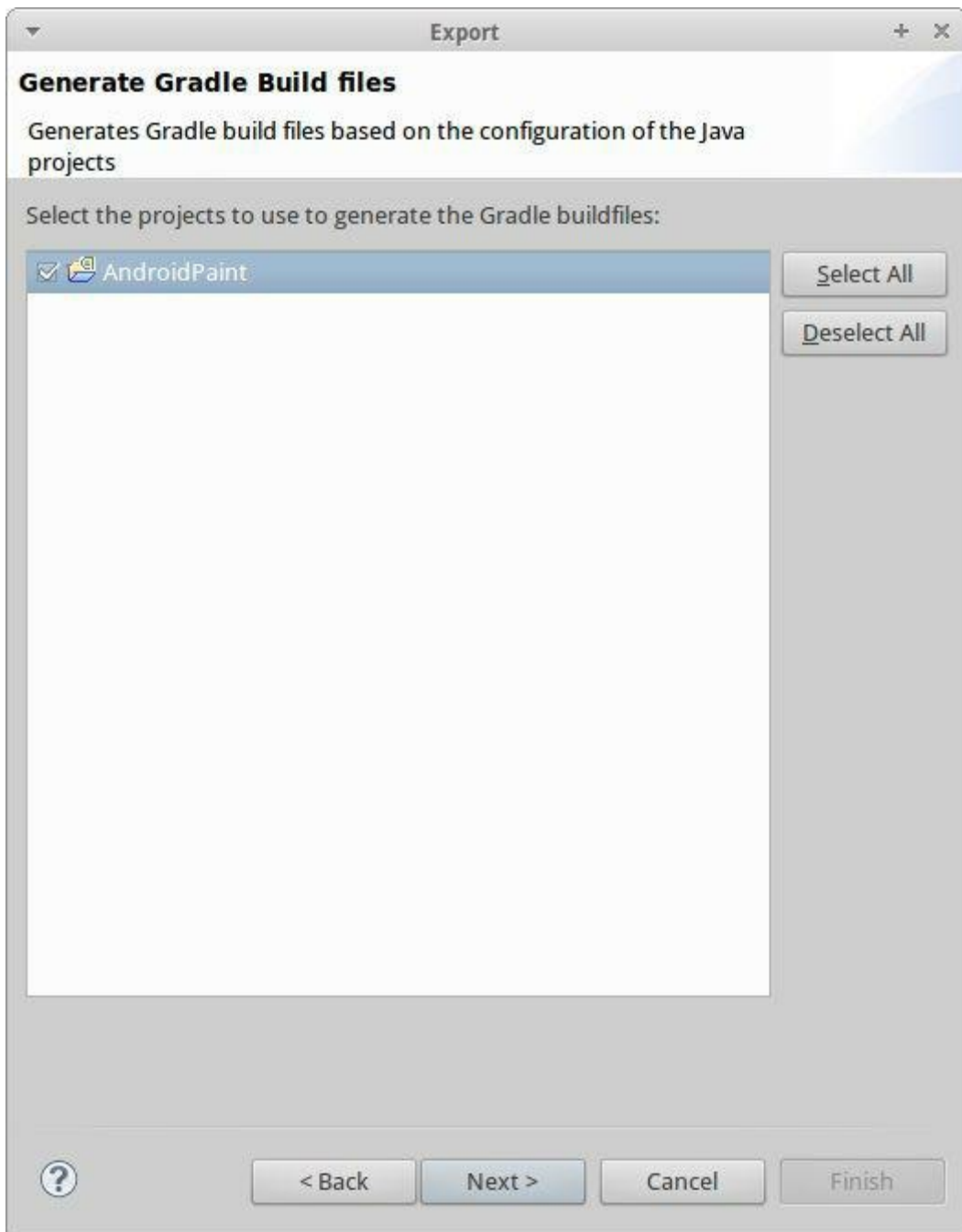
Name	API	Rev.	Status
Tools			
Android SDK Tools		22.0.5	Install
Android SDK Platform-tools		18.0.1	Install
Android SDK Build tools		18.0.1	Install

Open the project in Eclipse you like to use later in Android Studio.

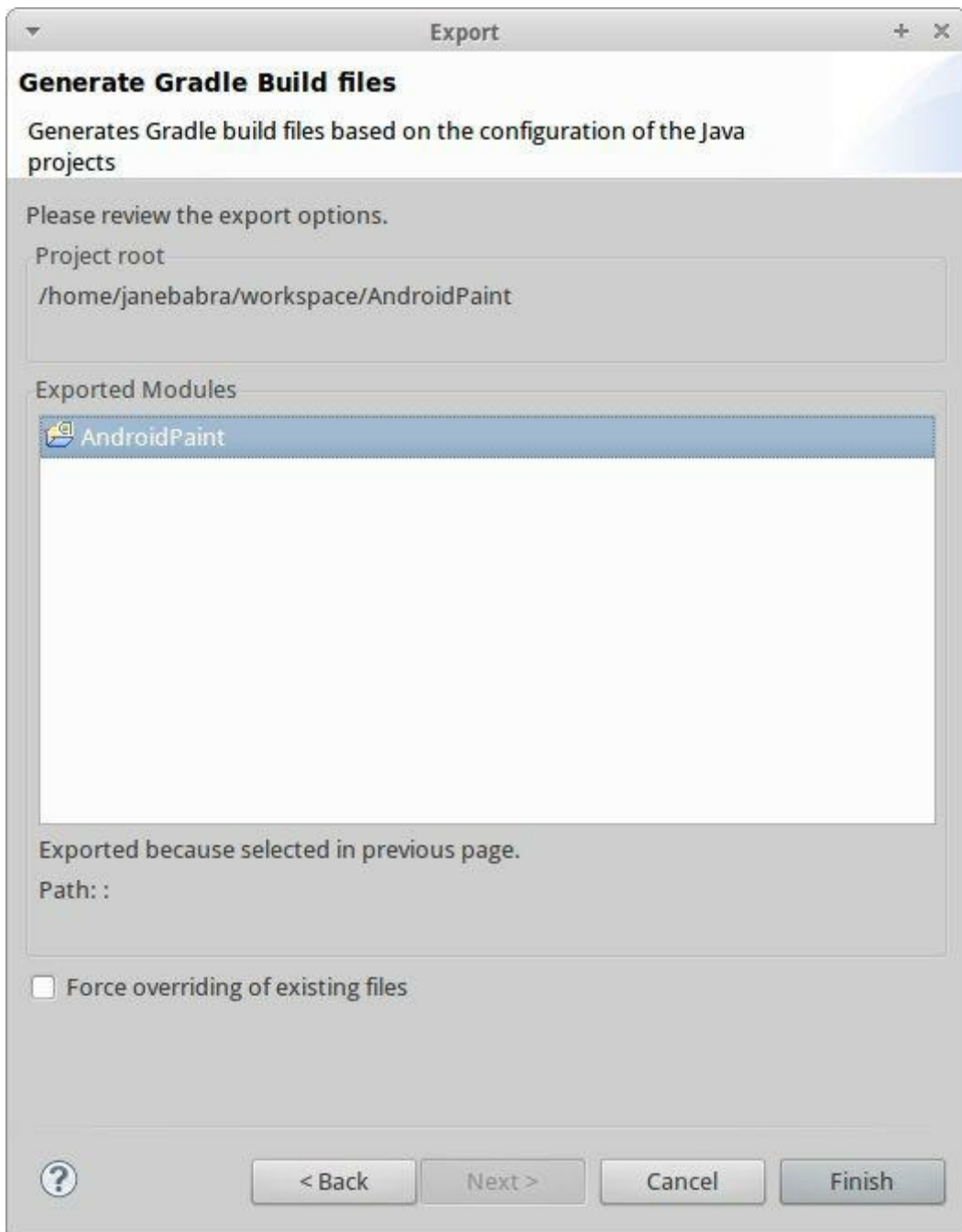
Then go to File -> Export ..



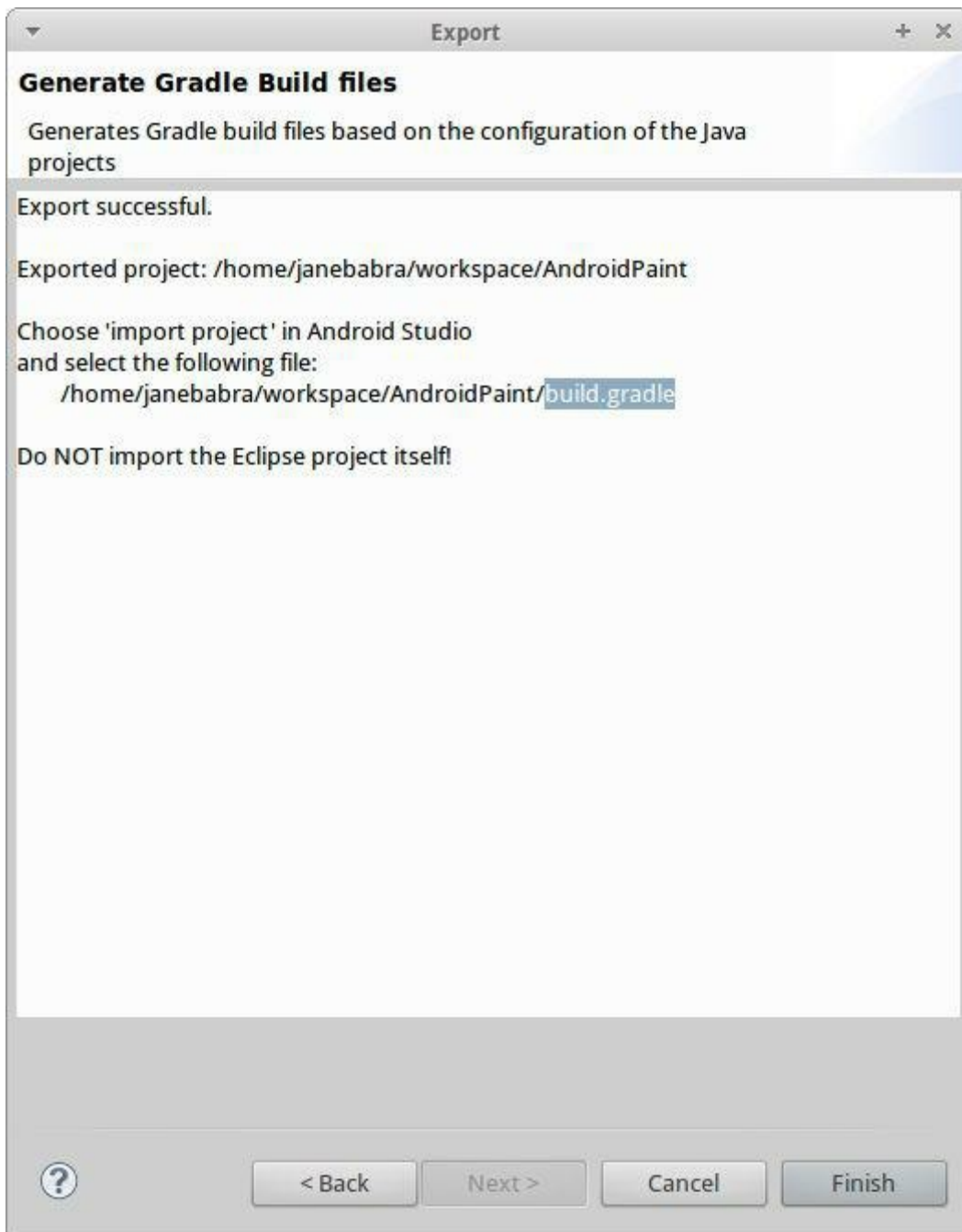
mark the "Generate Gradle build files" and continue with <Next>.



Select the project(s) from the list and press <Next> and <Finish>

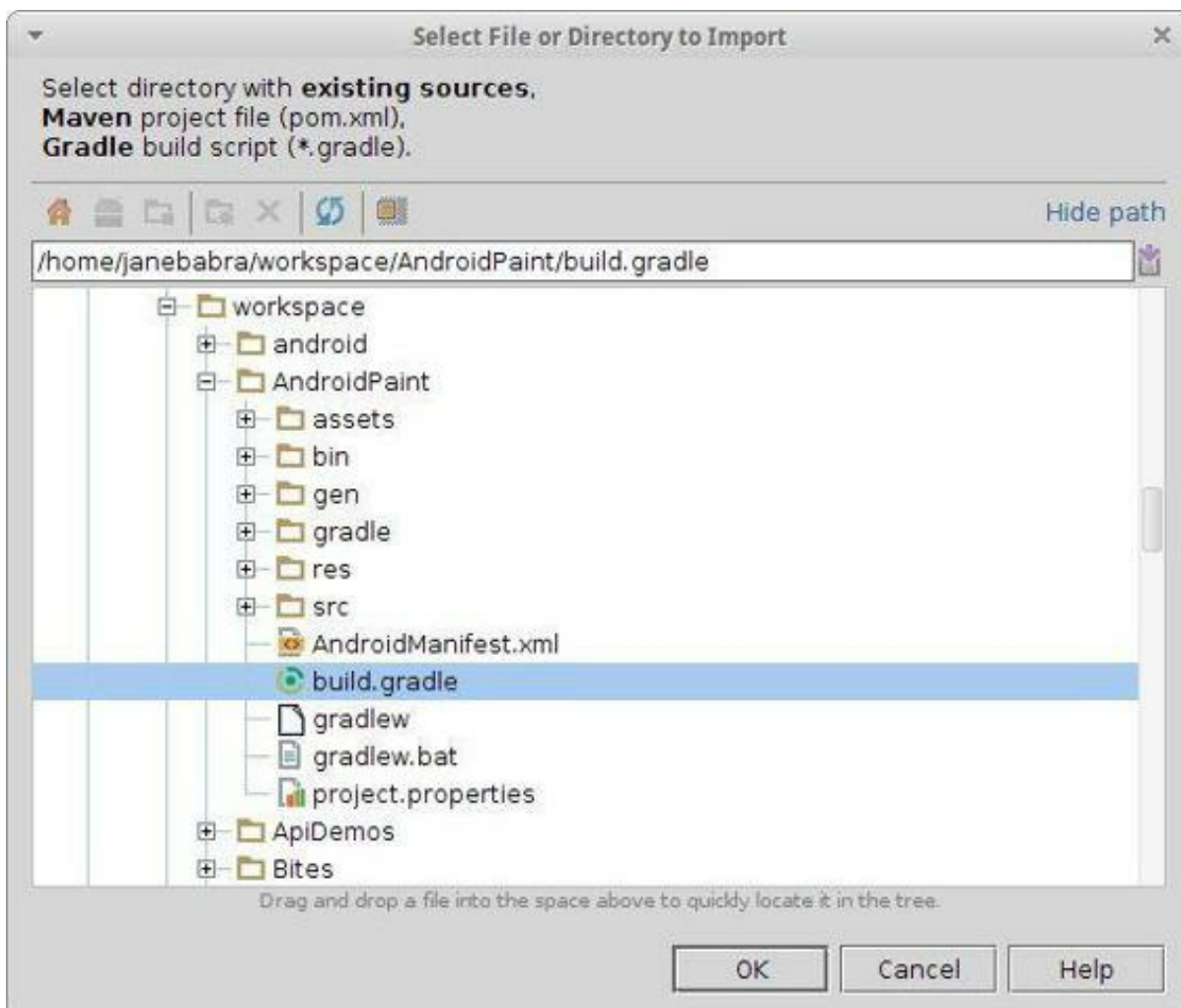


The "Export successful" window explains how to import this project later.

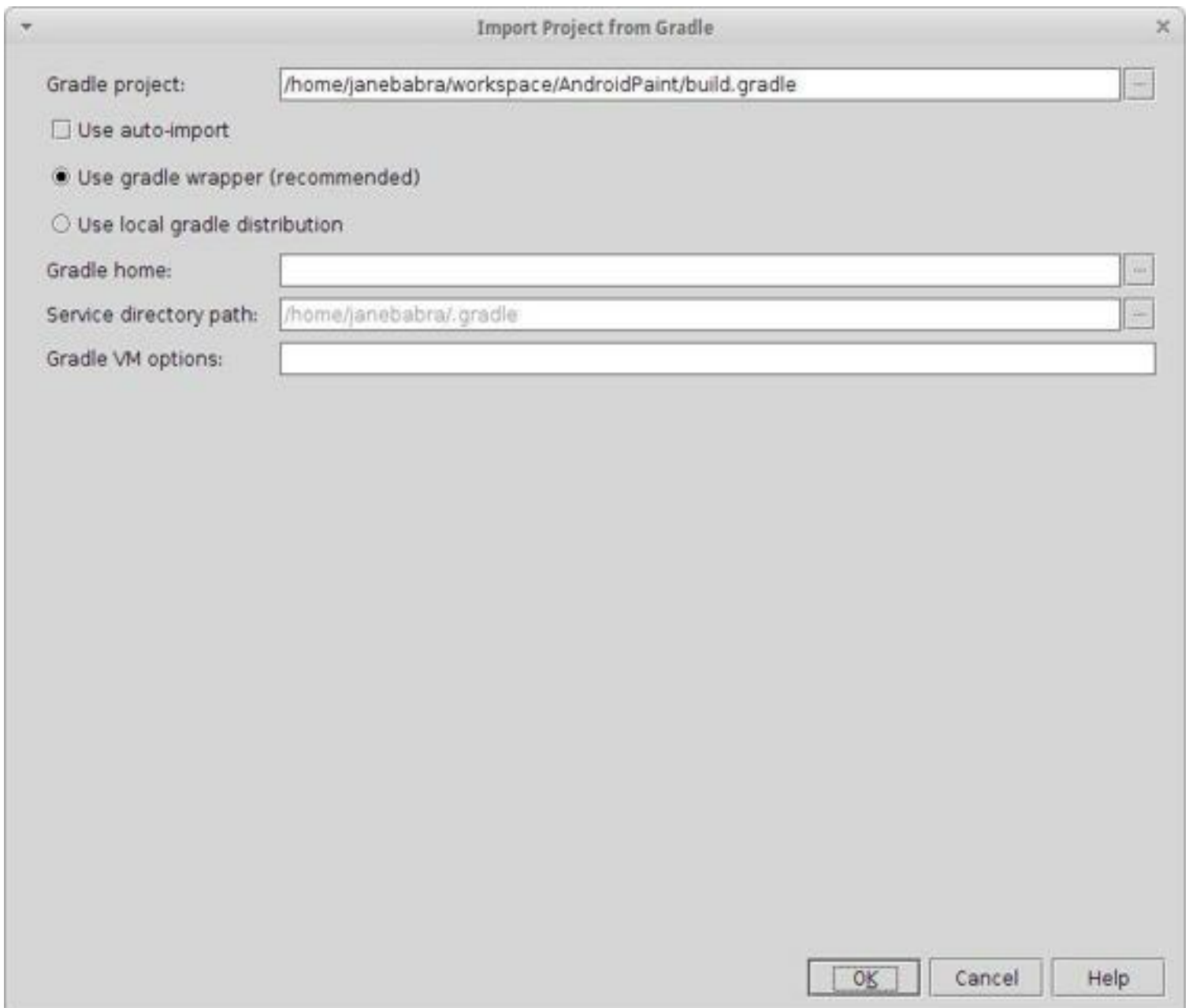


<Finish> and close Eclipse.

Open Android Studio, go from the Welcome Screen to Import Project and navigate to build.gradle file of the project. <OK>



Confirm the next screen "Import Project from Gradle" with <OK>. Don't change anything.



From now on the project can be developed in Android Studio.

Right now there're a lot of different eclipse/adt versions out there and several version Android Studio.

Here's an overview that maybe help when the procedure from this chapter is not working right away.

For Android Studio up to version 0.2

If you are using an Android Studio version before 0.2, you're done.

You must close Eclipse before opening Android Studio. From the Main Menu -> Import

Project

Confirm the suggested standard and finish importing.

For Android Studio from 0.2+ up to 0.2.13

If you're using Android Studio version 0.2 or higher, there's some manual editing to do:

When you have finished the export in Eclipse, open the new added file build.gradle.

Navigate to the following lines.

```
dependencies {  
  
    classpath 'com.android.tools.build:gradle:0.4'
```

and change the 0.4 into 0.5+ like this

```
dependencies {  
  
    classpath 'com.android.tools.build:gradle:0.5+'
```

This is necessary because Android Studio from version 0.2 works now with the Gradle plugin 0.5 and the Eclipse plugin hasn't been updated.

You must close Eclipse before opening Android Studio. From the Main Menu -> Import Project

Confirm the suggested standard and finish importing.

For Android Studio version 0.3+

Importing projects from Eclipse into Android Studio 0.3+ is right now a kind of pain in the ...

What I found out is, that the Android Studio versions 0.3.1 on Windows and Mac OS X are behaving very different.

Could not import in the Mac installation but in the Windows installation. Here's what I did:

Because this was an old Eclipse project without Gradle, I exported it from Eclipse to add the

gradle structure.

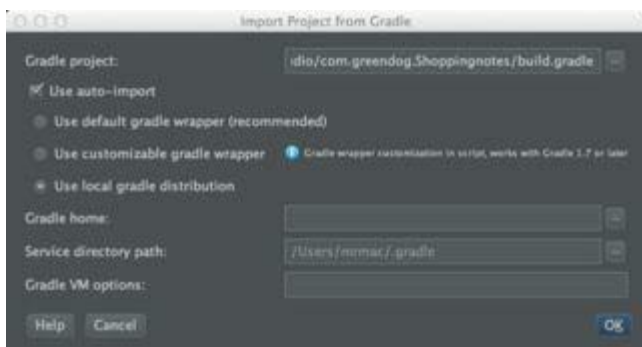
I updated the Gradle configuration to make it fit for Android Studio 0.3+

changed in build.gradle the gradle plugin version to 0.6+

copied from a project I made with 0.3 from scratch, the content of the wrapper folder into the projects wrapper folder

You must close Eclipse before opening Android Studio. From the Main Menu -> Import Project

Imported into Android Studio like this



Mac problem solved

After renaming .gradle in my home directory into .gradleold, I only had to change in build.gradle the gradle plugin version into 0.6+ . Android Studio was now able to download everything needed.

This is an ongoing story, Android Studio is still in development.

Import Android Studio projects like the "Samples" from the sdk in Android Studio

Welcome Screen -> "Import Project"

navigate to one of the samples directory like:

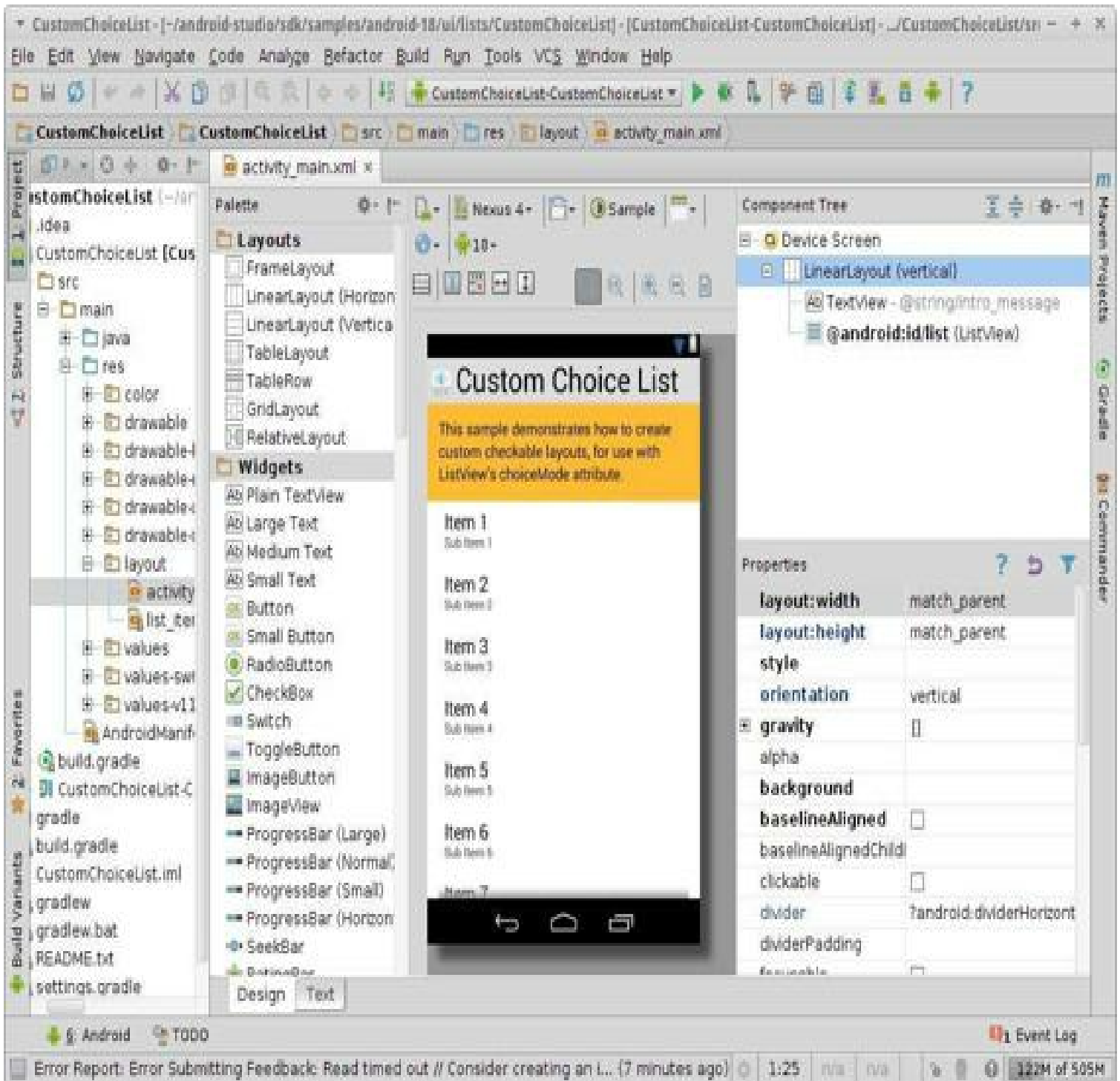
android-studio/sdk/samples/android-18/

connectivity, content, input, media, security, testing or ui.

Look for projects with Gradle files.

Under the folder "legacy" are old Eclipse Projects.

If you don't see any sample folder, install the samples via SDK Manager.



Import a VCS Project

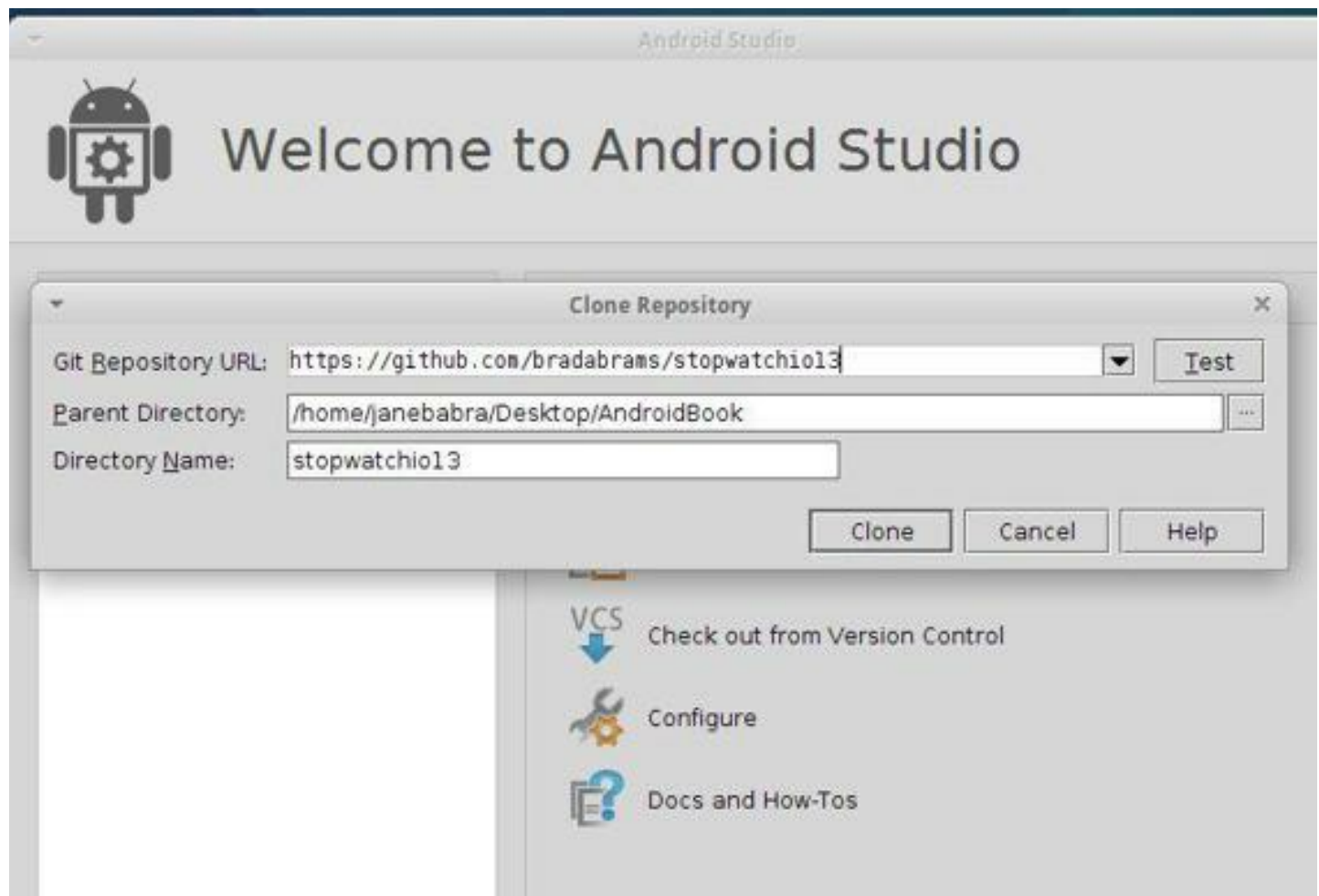
This could be your own project, from your team or a public project on for example GitHub code.google.com.

The project needs to have the Gradle build system.

Example for Git:

The project "StopWatch" will be imported. The project Google used to present Android Studio.

Welcome Screen -> "Checkout from Version Control"



Google Cloud Endpoints

Add to an Android Studio Project Google Cloud Endpoints.

Download Maven

<http://maven.apache.org/download.html>

Installation von Maven

Unpack the downloaded file.

Add the PATH to your environment like this:

Linux und Mac OS X:

```
export PATH = /usr/local/apache-maven-3.x.y/bin: $PATH
```

Windows:

```
set path = "c:\program files\apache-maven-3.x.y\bin";%PATH%
```

Make sure, you have JAVA_HOME set to the location of your JDK.

Test Maven

"mvn - version" checks the installation

In the Google Cloud Console: <https://cloud.google.com/console>

1. "Create a new project" and make a note of the "Project ID".
2. From Apis select and activate the "Google Cloud Messaging for Android API".
3. In the left corner click on "Register App".
4. Give the App a name.
5. Next select "Android" and "Accessing APIs via a webserver".
6. Click on <Register> to finish this part.

In the next screen

1. Expand the "Server Key" box.
2. Copy the API KEY in a text file to use it later in Android Studio.

3. Leave the Google Cloud Console.

Back to Android Studio

1. Create a new Android app project or open one.
2. In the project view panel go to the root of the Android App project
3. Menu Tools -> "Google Cloud Endpoints -> Create App Engine Backend"
4. In the "Generate App Engine Backend" window write all the information from the "Google Cloud Console".
5. Click on <create> to create the Backend.

Adding the GCM registration to the App

The RegisterActivity can be placed anywhere, for example with the onCreate ()-method.

Deploy the sample backend server

When you're ready to deploy an update to your (the sample) production backend in the cloud, you can do that easily from the IDE. Click on the "Maven Projects" button on the right edge of the IDE, under Plugins > App Engine, right-click and run the appengine:update goal.

As soon as the update is deployed, you can also access your endpoints through the APIs Explorer at http://<project-id>.appspot.com/_ah/api/explorer.

For testing and debugging, you can also run your backend server locally without having to deploy your changes to the production backend. To run the backend locally, just set the value of LOCAL_ANDROID_RUN to true in CloudEndpointUtils.java in the App Engine module.

Build and run the Android app

Now build and run your Android app. If you called RegisterActivity from within your main activity, the device will register itself with the GCM service and the App Engine app you just deployed. If you are running the app on an emulator, note that GCM functionality requires the Google APIs Add-on image, which you can download from the SDK Manager.

You can access your sample web console on any browser at <http://<project-id>.appspot.com>. There, you will see that the app you just started has registered with the backend. Fill out the form and send a message to see GCM in action!

Extending the basic setup

It's easy to expand your cloud services right in Android Studio. You can add new server-side code and through Android Studio instantly generate your own custom endpoints to access those

services from your Android app.

A good example for the Google Cloud Endpoint is the

[The App Example for Android Studio from the I/O Google](#)

Explains the Example

<http://bradabrams.com/2013/06/google-io-2013-demo-android-studio-cloud-endpoints-synchronized-stopwatch-demo/>

Google Play Service SDK

The Google Play Service SDK is a new Library. It includes for example Services like Google Maps Android API v2, Google+, Google In-app Billing v3 and Google Cloud Messaging.

To use one of the Services included in Google Play Service SDK, you have to download the library via SDK Manager and configure it in *build.gradle*. The package called "Google Play services" and is located under Extras.

The following sample shows the Integration of Google Maps API v2.



Integration of Google Maps API v2

Creating a SHA-1 fingerprint

Creating a SHA-1 fingerprint (release or debug Certificate) is necessary to get the Google Map v2 API Key.

Command for a Debug Certificate

Windows:

```
keytool -list -v -keystore "C:\Users\your_user_name\.android\debug.keystore" -alias androiddebugkey -storepass android -keypass android
```

Mac OS X und Linux:

```
keytool -list -v -keystore ~/.android/debug.keystore -alias androiddebugkey -storepass android -keypass android
```



Command for a Release Certificate

```
keytool -list -keystore your-keystore-name
```

The placeholder *your-keystore-name* has to be replaced with the full path and name of your own release keystore.

After confirming with ENTER, you will be prompted to enter your password, then you should see the aliases associated with the keystore.

Copy the SHA-1 fingerprint. It will be used later to get the Google Map v2 API Key from the Google API Console.

Creating a project in the Google API Console

Google API Console:

<https://code.google.com/apis/console>

From the Drop-Down Menu select <create> and give your project a name.



Select Services from the list on the left of the APIs console. You should see a list of Google services, scroll down to Google Maps Android API V2 and click to turn it on.

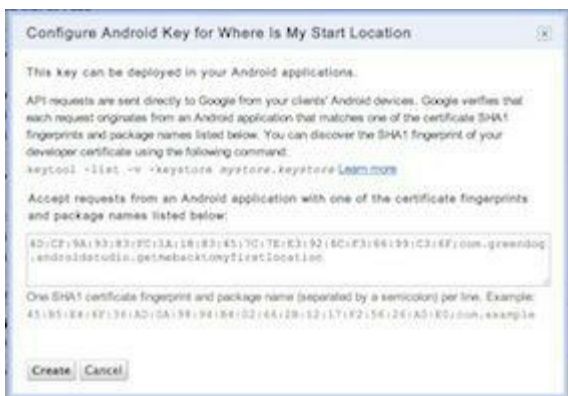


Go back to the menu to select **<API access>**. From here you will get the Google Map v2 API key.

Scroll down to and click on **<Create new Android key>**.



In the pop-up window that appears, enter the SHA-1 fingerprint you copied from your certificate add after a semicolon your package name and click Create.

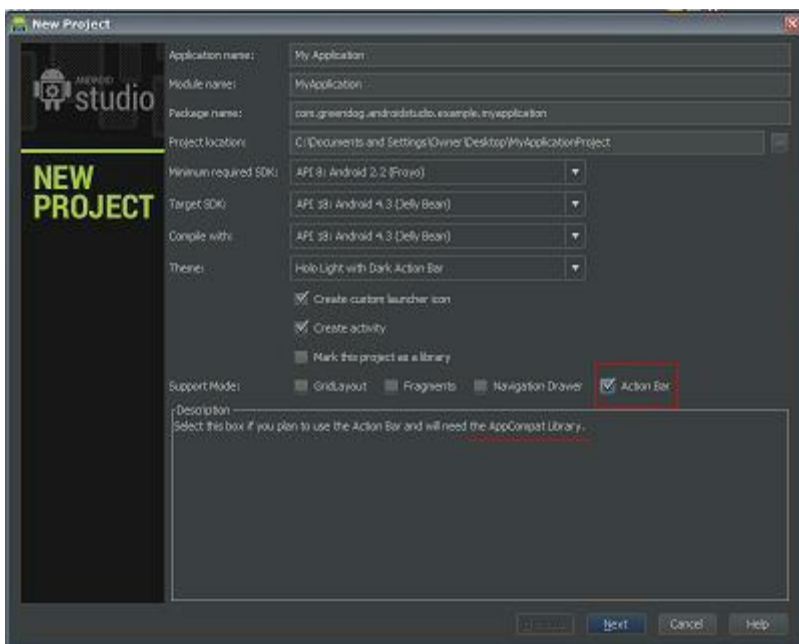


This will update your API Access side and you will find under **"Key for Android apps (with certificates)"** the API key. Copy the API key, so you can use it later in your *AndroidManifest.xml*.

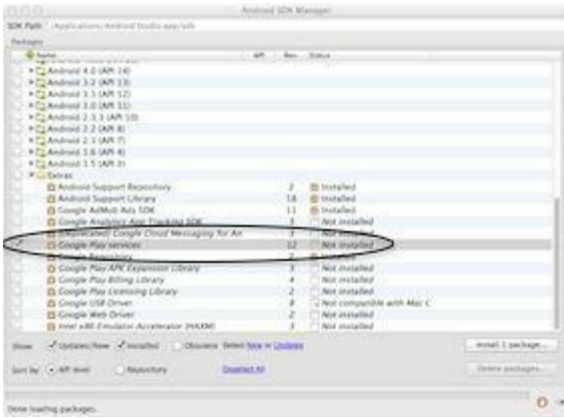


Creating a new Android Map Project

In the NEW PROJECT Window select from Support Mode **Fragments**. This make your live easier because Google Maps v2 are displayed in Fragments only. By clicking "Support Mode **Fragments**", Android Studio will add the dependencies in build.gradle for you. After this, continue with the creating and finish the process like you normally do.



Open SDK Manager and install Google Play Service



Edit the Gradle File

Open the *build.gradle* file and add the dependencies for Google Play Service:

```
dependencies {
    compile 'com.android.support:support-v4:18.0.0'
    compile 'com.android.support:appcompat-v7:18.0.0'
    compile 'com.google.android.gms:play-services:3.1.36'
}
```

Editing the AndroidManifest File

Open the *AndroidManifest.xml* and add the following:

```
<meta-data
    android:name="com.google.android.maps.v2.API_KEY"
    android:value="your_key_here" />
```

android:value="your-key-here"
put here your own API Key, and only here.

```
<permission
    android:name="your.package.name.permission.MAPS_RECEIVE"
    android:protectionLevel="signature" />
<uses-permission
    android:name="your.package.name.permission.MAPS_RECEIVE"/>
```

replace your.package.name with your own

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission
    android:name="com.google.android.providers.gsf.permission.READ_GSERVICES" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"
```

```

/>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />

<uses-feature
    android:glEsVersion="0x00020000"
    android:required="true"/>

```

Add a Map to the App

Replace the content from *activity_main.xml* with something like this:

```

<fragment
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:name="com.google.android.gms.maps.SupportMapFragment"
    android:id="@+id/map"
    android:layout_below="@+id/title"
    android:layout_alignLeft="@+id/title"
    android:layout_alignParentRight="true"
    android:layout_above="@+id/update_button"
    map:cameraTilt="45" />

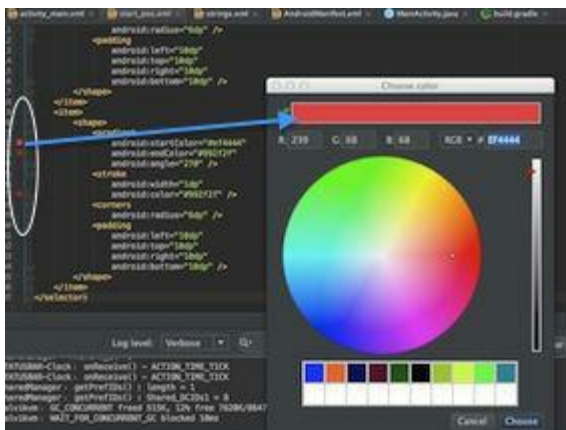
```

About

map:cameraTilt="45"

This is needed to make the compass visible. The compass is only visible when "tilt" or "bearing" is non-zero.

The button has been designed with *start_pos.xml* located in *drawable/*. If you open this file, you will see one of the new features from editor.



Clicking on the color icon on the left, opens the "Choose color" window, which makes it very easy to choose or change colors.

Java File

If you only show the map and don't do anything more, you can leave the *MainActivity.java* like it is.

For this example some code has been added to support for example "saving the current position", "add a marker", "show the compass and current position symbol" and "connect the

button".

The compass symbol has been activated through
`mMap.getUiSettings().setCompassEnabled(true);`

The current position symbol has been activated through
`mMap.setMyLocationEnabled(true);`

The complete Code for this project is on GitHub:

<https://github.com/janebabra/GetMeBackToMyFirstLocationProject>

In the build/ folder you will find apk's.

Product Flavors - Build Types - Build Variants

One of the goals of the new build system is to give the ability to create different app versions of the same app code.

The integration of Gradle as the new build system makes it possible to generate different APK's from the same project code base.

What are the main use cases?

1. To have different version of the same App, like a free and a paid version for example. This would properly be the most wanted use case.
2. To pack an app differently for an upload as multi-apk to the Google Play Store.
3. The combination of 1 and 2.

When building Flavors, the Flavors must have the same API.

Building the directory structure and making a `productFlavors` DSLcontainer entry in the project's `build.gradle` file create Flavors.

For the example, we are going to create an app with **two Flavors**, one called **production** and another one called **testing**.



The two Flavors will be different in

- App icon -> drawable-xxxx
- App name -> strings.xml
- Background color -> activity_main.xml
- Content -> Constants.java

Create a new Android Project and when it comes to the screen Activity name and Layout name, name both Layout names *activity_main.xml* because we're not using a Fragment Layout for this example.



Proceed like normal with the rest of the New Project Wizard.

The next task will be to build the directory structure. This has to be done manually in the Project Structure Window.

In the **src/** folder, you've to create two directories whose names must match the flavors. Then you can define all the flavor-specific values. Only specific values are necessary.

This is the complete directory structure for the project:

This is the complete directory structure for the project:

```
|— main
|  |— AndroidManifest.xml
|  |— ic_launcher-web.png
|  |— java
|  |  |— com
|  |  |  |— greendog
|  |  |  |  |— buildtypesflavors
|  |  |  |  |  |— MainActivity.java
|  |— res
|  |  |— drawable-hdpi
|  |  |  |— ic_launcher.png
|  |  |— drawable-mdpi
|  |  |  |— ic_launcher.png
|  |  |— drawable-xhdpi
|  |  |  |— ic_launcher.png
|  |  |— drawable-xxhdpi
|  |  |  |— ic_launcher.png
|  |  |— layout
|  |  |  |— activity_main.xml
|  |  |— menu
|  |  |  |— main.xml
|  |— values
```

```
| | └─ dims.xml
| | └─ strings.xml
| | └─ styles.xml
| └─ values-v11
| | └─ styles.xml
| └─ values-v14
|   └─ styles.xml
└─ production
  └─ java
    └─ com
      └─ greendog
        └─ buildtypesflavors
          └─ Constants.java
└─ testing
  └─ java
    └─ com
      └─ greendog
        └─ buildtypesflavors
          └─ Constants.java
└─ res
  └─ drawable-hdpi
    └─ ic_launcher.png
  └─ drawable-mdpi
```

```

|   └─ ic_launcher.png
├── drawable-xhdpi
|   └─ ic_launcher.png
├── drawable-xxhdpi
|   └─ ic_launcher.png
└─ values
    └─ string.xml

```

The build.gradle file gets the productFlavors entry inside the android section:

```

productFlavors {
    production {
        packageName "com.greendog.buildtypesflavors"
    }

    testing {
        packageName "com.greendog.buildtypesflavors.staging"
    }
}

```

Important:

The names production and testing have to match the names for the Flavors folder. They can't be different MainActivities, one in the main/ folder and another in one of the Flavor folders.

To make the different content in the testing Flavor, we need a second Java file to define this difference. In this example, the extra Java file is called Constants.java.

The Constans.java for the production Flavor:

```
package com.greendog.buildtypesflavors;
```

```
public class Constants {  
    public final static String BASE_URL = "https://github.com/janebabra?tab=repositories";  
}
```

The Constans.java for the testing Flavor:

```
package com.greendog.buildtypesflavors;
```

```
public class Constants {  
  
        public        final        static        String        BASE_URL        =  
"https://github.com/janebabra/MySwipeImageViewerProject/blob/master/README.md";  
  
}
```

For resource files, it's a bit different. You can have the same filenames like strings.xml in all of the Flavor folders and in the main folder.

The concept of Gradle is that you can define Build Types and Build Flavors for your project. The Build Types are Debug and Release. The Product Flavors in this example are production and testing. When combining Build Types and Product Flavors, you get Build Variants.

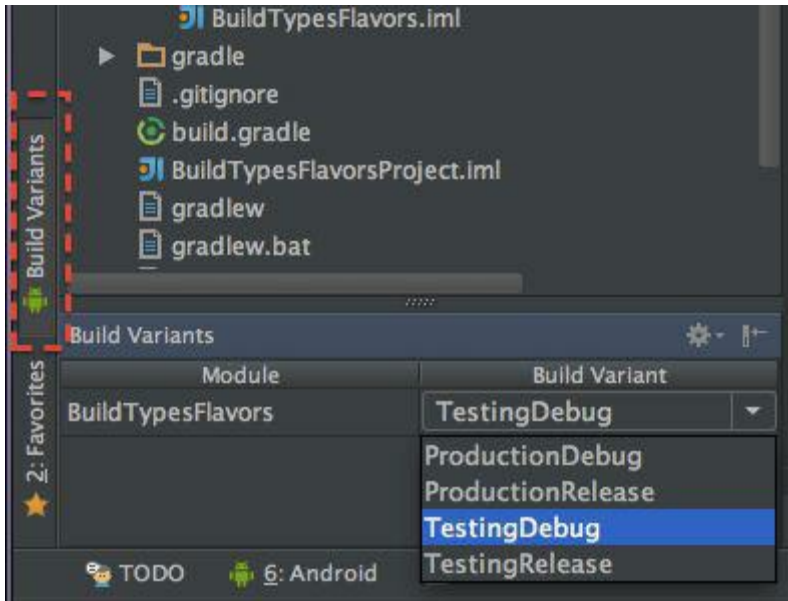
With this example you can build 4 different variants:

1. ProductionDebug
2. ProductionRelease
3. TestingDebug
4. TestingRelease

Running and building the different Build Variants

From the IDE

Click the <Build Variants> tab on the Left of the IDE to open the Build Variants Window.



The Drop-Down menu under Build Variant let you switch between the available Build Variants. Next click the <Run> Button. This will either install on a device or the Emulator.

The method works for the Debug Variants.

When choosing a Release Variant, the build.gradle file needs the entry for your Release key like the Gradle User Guide describes.

Example:

```
android {  
    signingConfigs {  
        production {  
  
storeFile file("other.keystore")  
        storePassword "android"  
        keyAlias "androiddebugkey"
```

```
keyPassword "android"
```

```
}
```

```
}
```

Building the different Build Variants from the command line

When Product Flavors are used, more assemble-type tasks are created. These are:

- `assemble<Variant Name>`
- `assemble<Build Type Name>`
- `assemble<Product Flavor Name>`

`gradlew assemble`

is building all for Build Variants

`gradlew assembleDebug`

will build `ProductionDebug` and `TestingDebug`

`gradlew assembleRelease`

will build `ProductionRelease` and `TestingRelease`

If you only need one of the Flavors, add the Flavors name, like

`gradlew assembleProductionDebug`

`gradlew assembleProduction`

will produce `ProductionDebug` and `ProductionRelease`

The Source Code for this project is located on GitHub:

<https://github.com/janebabra/BuildTypesFlavorsProject>

Game Development with Android Studio

This Chapter includes the Open Source Games Engines libGDX and AndEngine and the integration of JavaScript games based on LIME.

AndEngine

Preparing AndEngine

1. Download and import the library project into Eclipse first
<https://github.com/nicolasgramlich/AndEngine>
2. In Eclipse export the AndEngine project (without the AndroidManifest.xml)
File -> Export -> Java -> JAR file and save as AndEngine.jar
3. In Android Studio
 1. Start a new project called "TowerOfHanoi".
 2. Give your Activity the name "TowerOfHanoiActivity.java"
 3. give both xml files the same name to avoid the Fragment
 4. Create a libs/ folder in the root of the project and copy the AndEngine.jar into it
 5. Go to project structure
 6. Libraries
 7. Plus button > java > and search for jar.
 8. Assign the jar to your project
4. Open the build.gradle file and add

in "dependencies" add "compile files('libs/Andengine.jar')"

like this:

```
dependencies {  
  
    compile files('libs/android-support-v4.jar')  
  
    compile files('libs/AndEngine.jar')
```

DONE! AndEngine is ready.

Examples for AndEngine can be found here:

<https://github.com/nicolasgramlich/AndEngineExamples>

Before you start coding from scratch or importing one of the samples, there are a few things you should know about AndEngine:

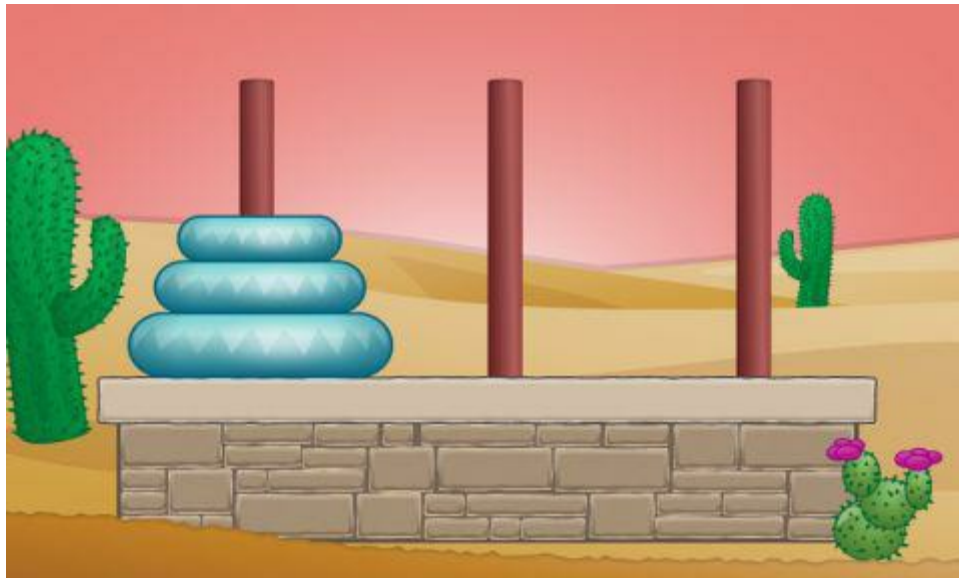
- **OpenGL ES 2.0:** The currently developed version of AndEngine is the GLES 2.0 version. This version requires OpenGL ES 2.0 support in order to function. So you need to make sure that your device (or emulator) supports OpenGL ES 2.0. If you are using the

emulator to test, then you need graphics acceleration support and this is there only in the SDK Tools higher than 19. In Android Studio has already a higher version.

- **Android 4.0.3:** You also need Android SDK Platform API 15, Revision 3 or higher in order to test OpenGL ES 2.0 based code on the emulator. So make sure that you've upgraded to at least Android 4.0.3, which is API 15.
- **Emulator:** If you are using the emulator to test AndEngine code, you need a virtual device which has **GPU emulation enabled** and is running at least Android 4.0.3. So check your virtual device configuration and edit it if your device is not set up appropriately.

Test Project for AndEngine: Tower of Hanoi

The objective of the Tower of Hanoi is to move all the rings over to the third rod, while only moving one ring at a time and without placing a larger ring on top of a smaller one.



Navigate to your main/java folder and open the “TowerOfHanoiActivity.java” file.

Instead of extending the **Activity** class, you want to make it extend a class called **SimpleBaseGameActivity**. And in order to refer to the SimpleBaseGameActivity class from the AndEngine Library, we need to add a few import statements for it. So we add that first to the top of the file under the existing import line:

```
import org.andengine.ui.activity.SimpleBaseGameActivity;  
import org.andengine.engine.options.EngineOptions;  
import org.andengine.entity.scene.Scene;
```

```
public class TowerOfHanoiActivity extends SimpleBaseGameActivity {
```

The SimpleBaseActivity class provides additional callbacks and contains the code to make AndEngine work with the Activity life cycle. Each callback that it provides is used for a specific purpose. As soon as you extend this class, you'll have to override three functions.

Here is a brief description of each of those functions:

- **onCreateEngineOptions:** This function is where you create an instance of the engine. Every activity that the game uses will have its own instance of the engine that will run within the activity lifecycle.
- **onCreateResources:** This is the function where you'll load all the resources that the activity requires into the the VRAM.
- **onCreateScene:** This function is called after the above two callbacks are executed. This is where you create the scene for your game and use all the textures that you previously loaded into memory.

We will first **add some empty placeholders** for the above callbacks so that we have some skeleton code. Replace the existing contents of our TowerOfHanoiActivity class with the following:

```
@Override
public EngineOptions onCreateEngineOptions () {
    // TODO Auto-generated method stub
    return null;
}

@Override
protected void onCreateResources () {
    // TODO Auto-generated method stub
}

@Override
protected Scene onCreateScene () {
    // TODO Auto-generated method stub
    return null;
}
```

Next, create a few static variables. These will be private to our class. So add the following code right below the class definition line:

```
private static int CAMERA_WIDTH = 800;
private static int CAMERA_HEIGHT = 480;
```

These two static variables define the width and height of the camera that the engine will use. This means that the final dimensions of the game scene will be equal to the camera size and width.

Next, you're going to initialize an instance of the engine. But that code is going to require several import statements in order to function properly. So first add the following imports to the top of the file below the existing import statements:

```
import org.andengine.engine.camera.Camera;
import org.andengine.engine.options.ScreenOrientation;
import org.andengine.engine.options.resolutionpolicy.RatioResolutionPolicy;
```

Now, add the following code for the **onCreateEngineOptions** function, replacing the placeholder content which is in there at the moment:

```
final Camera camera = new Camera(0, 0, CAMERA_WIDTH, CAMERA_HEIGHT);
return new EngineOptions(true, ScreenOrientation.LANDSCAPE_FIXED,
    new RatioResolutionPolicy(CAMERA_WIDTH, CAMERA_HEIGHT), camera);
```

In the above code, you create a new instance of the Camera class. Then we use that camera object to create the EngineOptions object that defines the options with which the engine will be initialized. The parameters that are passed while creating an instance of EngineOptions are:

- **FullScreen:** A boolean variable signifying whether or not the engine instance will use a fullscreen.
- **ScreenOrientation:** Specifies the orientation used while the game is running.
- **ResolutionPolicy:** Defines how the engine will scale the game assets on phones with different screen sizes.
- **Camera:** Defines the width and height of the final game scene.

Android runs on a lot of devices with different screen sizes. AndEngine comes with a unique solution to this problem: it will automatically scale the game assets to fit the screen size of the device.

If you set your CAMERA_WIDTH/CAMERA_HEIGHT to 480×320 and someone runs it on a phone with a 800×480 screen size, your game will be scaled up to 720×480 (1.5x) with a 80px margin (top, bottom, left, or right). Notice that AndEngine keeps the same aspect ratio and scales the game scene to the closest possible value to the actual screen size.

Loading Game Assets to VRAM

Now that you've initialized an instance of the engine, you can load all the assets required by the Tower of Hanoi game into memory. **First download the game assets from here:**

<https://github.com/janebabra/TowerofHanoiProject>

Next create the folder named src/main/assets/**gfx/**. After that, copy all the downloaded assets to that folder.

To load these assets, we're going to add the **onCreateResources** method. Add to your imports the following new import statements:

```
import org.andengine.opengl.texture.ITexture;
import org.andengine.opengl.texture.bitmap.BitmapTexture;
import org.andengine.util.adt.io.in.IInputStreamOpener;
import org.andengine.util.debug.Debug;

import java.io.IOException;
import java.io.InputStream;
```

Now, replace the placeholder content in **onCreateResources** with the following:

```

try {
    // 1 - Set up bitmap textures
    ITexture backgroundTexture = new BitmapTexture(this.getTextureManager()
        @Override
        public InputStream open() throws IOException {
            return getAssets().open("gfx/background.png");
        }
    });
    ITexture towerTexture = new BitmapTexture(this.getTextureManager(), new
        @Override
        public InputStream open() throws IOException {
            return getAssets().open("gfx/tower.png");
        }
    });
    ITexture ring1 = new BitmapTexture(this.getTextureManager(), new IInput
        @Override
        public InputStream open() throws IOException {
            return getAssets().open("gfx/ring1.png");
        }
    });
    ITexture ring2 = new BitmapTexture(this.getTextureManager(), new IInput
        @Override
        public InputStream open() throws IOException {
            return getAssets().open("gfx/ring2.png");
        }
    });
    ITexture ring3 = new BitmapTexture(this.getTextureManager(), new IInput
        @Override
        public InputStream open() throws IOException {
            return getAssets().open("gfx/ring3.png");
        }
    });
    // 2 - Load bitmap textures into VRAM
    backgroundTexture.load();
    towerTexture.load();
    ring1.load();
    ring2.load();
    ring3.load();
} catch (IOException e) {
    Debug.e(e);
}

```

In the above code, you first create an ITexture object. ITexture is an interface. An object of this type is initialized to a BitmapTexture object, which is used to load a bitmap into VRAM. The above code creates ITexture objects for all the assets you downloaded, and loads them into VRAM by calling the load method on each object.

Now that you have all your assets loaded, you need to extract TextureRegions from your textures.

Note: Instead of creating textures for each of your assets, you could have loaded all the

assets into one texture and extracted the individual assets as TextureRegions. But this is not part of this example.

To handle the work with the TextureRegions we have to add a few new import statements:

```
import org.andengine.opengl.texture.region.ITextureRegion;
import org.andengine.opengl.texture.region.TextureRegionFactory;
```

Now, to hold the TextureRegions, add private variables to our class at the top of the file, below the previous private variables:

```
private ITextureRegion mBackgroundTextureRegion, mTowerTextureRegion, mRing
```

Finally, add these lines of code to **onCreateResources**, right after the end of section #2 where you load the bitmap textures into VRAM:

```
// 3 - Set up texture regions
this.mBackgroundTextureRegion = TextureRegionFactory.extractFromTexture(bac
this.mTowerTextureRegion = TextureRegionFactory.extractFromTexture(towerTex
this.mRing1 = TextureRegionFactory.extractFromTexture(ring1);
this.mRing2 = TextureRegionFactory.extractFromTexture(ring2);
this.mRing3 = TextureRegionFactory.extractFromTexture(ring3);
```

The above code initializes your TextureRegions using the textures that you already loaded into VRAM.

Creating the Game Scene

It's finally time to create the game scene! Now, we need an import for the sprites:

```
import org.andengine.entity.sprite.Sprite;
```

Next, replace the placeholder content in **onCreateScene** with the following:

```
// 1 - Create new scene
final Scene scene = new Scene();
Sprite backgroundSprite = new Sprite(0, 0, this.mBackgroundTextureRegion, c
scene.attachChild(backgroundSprite);
return scene;
```

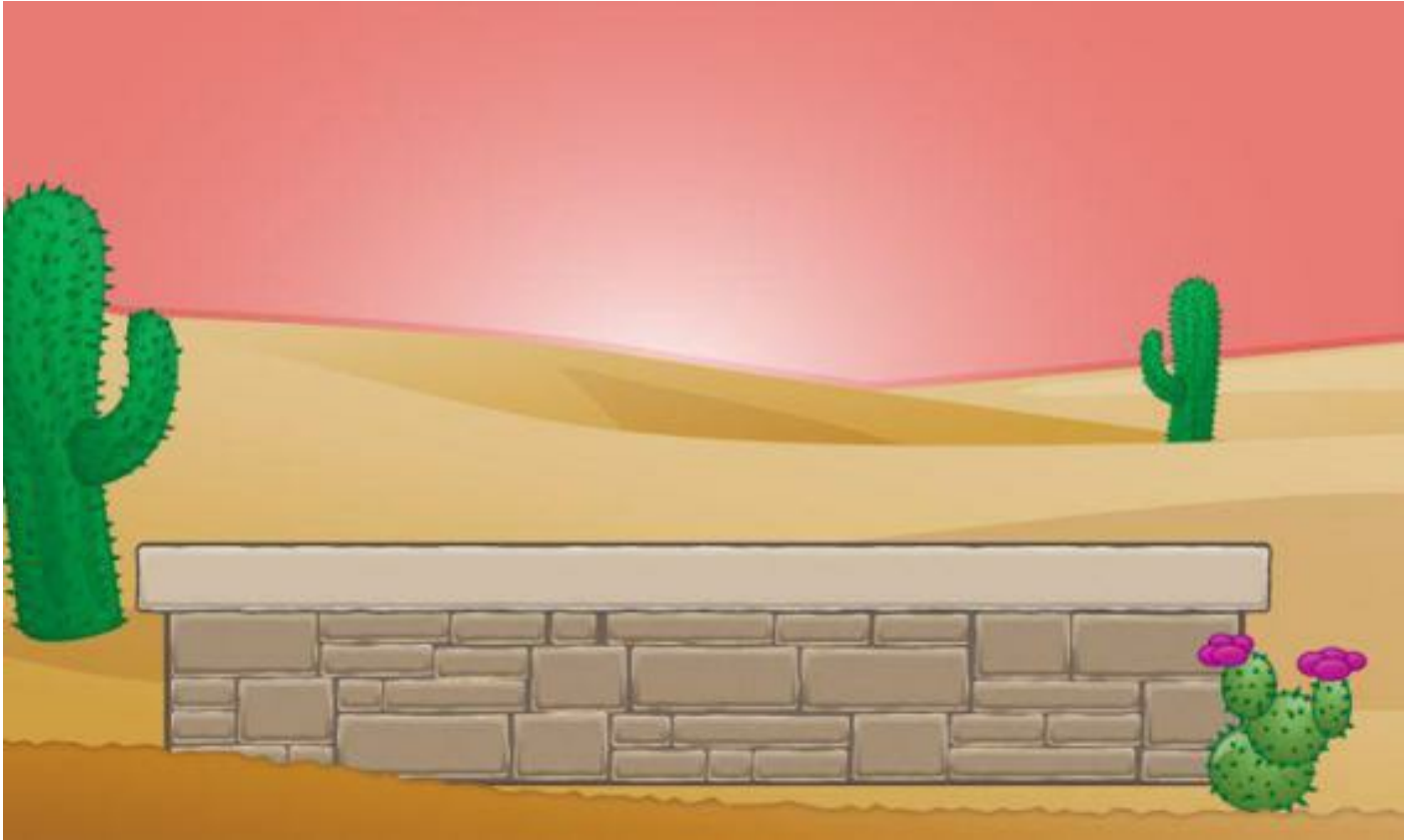
The above code first creates a Scene object. Next you create a sprite called backgroundSprite and attach the sprite to the scene. This method requires you to return the scene object.

When creating a Sprite object, you pass four parameters. Here's a brief description of each parameter:

- **xCoordinate**: Defines the X-position of the sprite. The AndEngine coordinate system considers the top-left point as the origin.
- **yCoordinate**: Defines the Y-position of the sprite.
- **TextureRegion**: Defines what part of the texture the sprite will use to draw itself.
- **VertexBufferObjectManager**: Think of a vertex buffer as an array holding the coordinates of a texture. These coordinates are passed to the OpenGL ES pipeline and

ultimately define what will be drawn. A `VertexBufferObjectManager` holds all the vertices of all the textures that need to be drawn on the scene.

Compile and run the application. You should see the loaded backgroundSprite:



The Three Towers

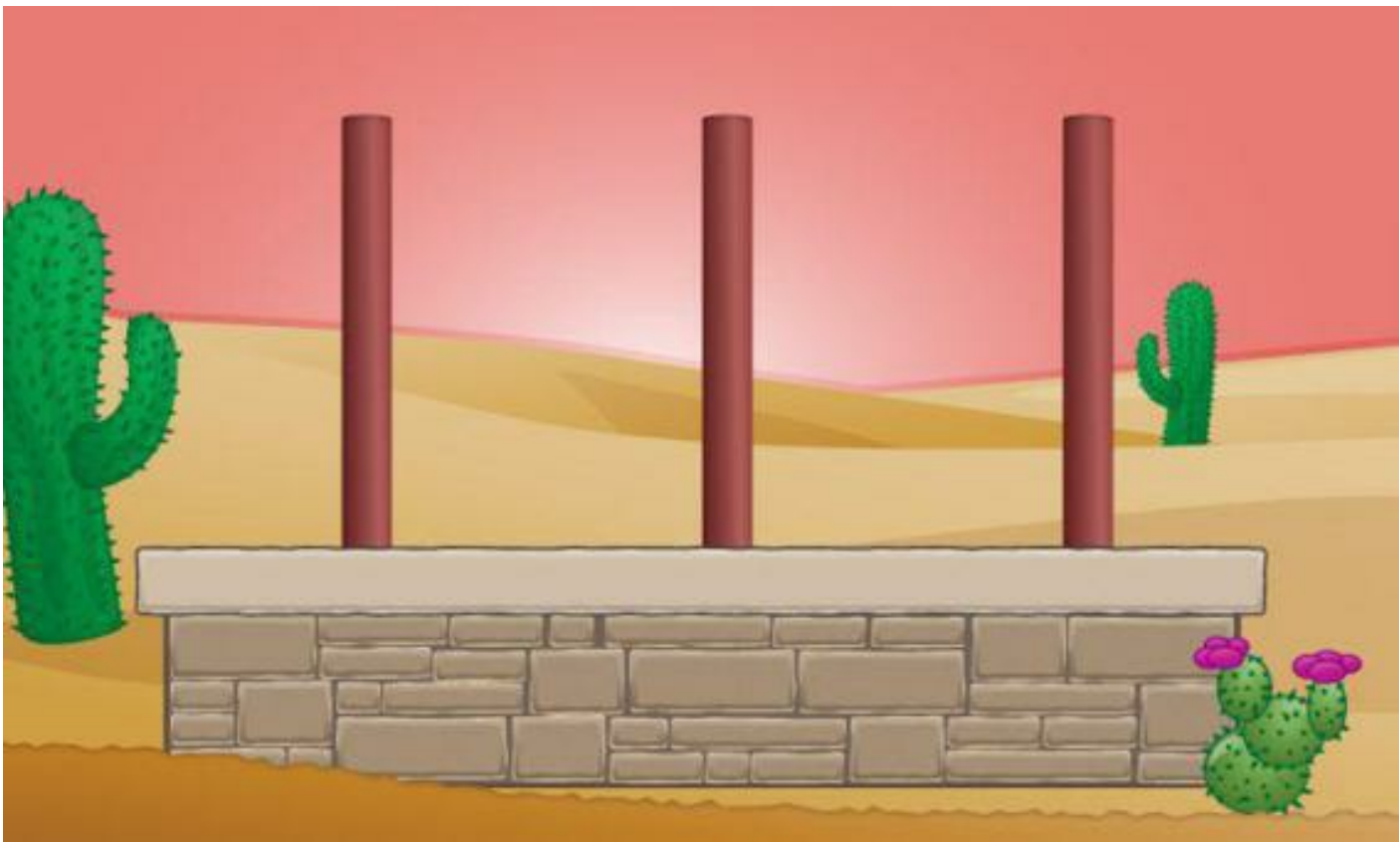
It's time to add the sprites for the towers and rings – the final step before we start with the game logic. Add three private variables to the class and declare the variables as follows:

```
private Sprite mTower1, mTower2, mTower3;
```

Next add the following lines of code to `onCreateScene`, right before the final return statement:

```
// 2 - Add the towers
mTower1 = new Sprite(192, 63, this.mTowerTextureRegion, getVertexBufferObjectManager());
mTower2 = new Sprite(400, 63, this.mTowerTextureRegion, getVertexBufferObjectManager());
mTower3 = new Sprite(604, 63, this.mTowerTextureRegion, getVertexBufferObjectManager());
scene.attachChild(mTower1);
scene.attachChild(mTower2);
scene.attachChild(mTower3);
```

You've defined three sprites, each using the `TextureRegion` of the tower that you loaded in `onCreateResources`. Then you added these sprites to your scene. That's all there is to it! Compile and run. You should now see the three towers placed in their proper positions.



And One Ring to Bind Them

Let's talk a little about the game logic before you create your rings. Think of the towers as three stacks (I mean the data structure) – you can only remove the top-most element, and when you add an element it will always be on top. You'll use these stacks when you write the game logic code.

To create the rings, we need to first make a custom class that will extend `Sprite`. You do this because every ring needs to know which stack it belongs to.

Right-click on the folder containing `TowerOfHanoiActivity.java` and select

New -> Class. Name the Class “`Ring.java`.” and place the following code within the class implementation (after the public class line and before the closing curly brace):

```
private int mWeight;
private Stack mStack; //this represents the stack that this ring belongs to
private Sprite mTower;

public Ring(int weight, float pX, float pY, ITextureRegion pTextureRegion) {
    super(pX, pY, pTextureRegion, pVertexBufferObjectManager);
    this.mWeight = weight;
}

public int getmWeight() {
    return mWeight;
}
```

```

public Stack getmStack() {
    return mStack;
}

public void setmStack(Stack mStack) {
    this.mStack = mStack;
}

public Sprite getmTower() {
    return mTower;
}

public void setmTower(Sprite mTower) {
    this.mTower = mTower;
}

```

Most of the code here is pretty straightforward. The object has three private variables. One is used to keep track of the weight of the tower; this is an integer value, i.e., the higher the value, the bigger the ring. The other two variables are used to store the stack that the ring belongs to and the tower on which it is currently placed.

You'll also need to add the following import statements to the top of the file:

```

import java.util.Stack;
import org.andengine.opengl.texture.region.ITextureRegion;
import org.andengine.opengl.vbo.VertexBufferObjectManager;

```

Now that we have the Ring class, to create and add the rings, add the following lines of code to **onCreateScene**, right before the return statement:

```

// 3 - Create the rings
Ring ring1 = new Ring(1, 139, 174, this.mRing1, getVertexBufferObjectManager);
Ring ring2 = new Ring(2, 118, 212, this.mRing2, getVertexBufferObjectManager);
Ring ring3 = new Ring(3, 97, 255, this.mRing3, getVertexBufferObjectManager);
scene.attachChild(ring1);
scene.attachChild(ring2);
scene.attachChild(ring3);

```

Compile and run to test.

You'll notice that the rings are now on the first tower but you can't move the rings. That's because we haven't worked out the game logic for placing and moving the rings.

Game Logic

Ready to bring your Tower of Hanoi puzzle to life? As mentioned before, as the foundation of the game logic, you're going to create three stacks, each representing a tower. Start by adding the following import for the Stack class to **TowerOfHanoiActivity.java**:

```

import java.util.Stack;

```

Next, declare the stack variables below the other private variables:

```

private Stack mStack1, mStack2, mStack3;

```

You'll initialize these variables in **onCreateResources**. Add the following lines of code after the end of section #3:

```
// 4 - Create the stacks
this.mStack1 = new Stack();
this.mStack2 = new Stack();
this.mStack3 = new Stack();
```

When the game starts, all three rings should be in the first stack. Put the following code in **onCreateScene** right before the return statement:

```
// 4 - Add all rings to stack one
this.mStack1.add(ring3);
this.mStack1.add(ring2);
this.mStack1.add(ring1);
// 5 - Initialize starting position for each ring
ring1.setmStack(mStack1);
ring2.setmStack(mStack1);
ring3.setmStack(mStack1);
ring1.setmTower(mTower1);
ring2.setmTower(mTower1);
ring3.setmTower(mTower1);
// 6 - Add touch handlers
scene.registerTouchArea(ring1);
scene.registerTouchArea(ring2);
scene.registerTouchArea(ring3);
scene.setTouchAreaBindingOnActionDownEnabled(true);
```

In the above code, you do the following:

1. Added the rings to the first stack.
2. Set the stack variable of each ring as the first stack and the tower variable as the first tower.
3. To handle touch and movement of the rings, you registered each ring as a touchable area.
4. Enabled touch binding.

Now, you need to override the **onAreaTouch** method of the Sprite class. This is where you'll add logic to move the rings. But that code in turn will require a method which checks whether a ring collided with a tower. You'll write that code later, but you need to add an empty method place holder for collision detection as follows (you can add this to the end of the class):

```
private void checkForCollisionsWithTowers(Ring ring) {
}
```

You also need to add the following import in order for the ring movement code to be able to identify the relevant classes:

```
import org.andengine.input.touch.TouchEvent;
```

Now, replace the first three lines of section #3 in **onCreateScene** (where you defined the

rings) with the following:

```
Ring ring1 = new Ring(1, 139, 174, this.mRing1, getVertexBufferObjectManager)
@Override
public boolean onAreaTouched(TouchEvent pSceneTouchEvent, float pTouchX, float pTouchY) {
    if (((Ring) this.getmStack().peek()).getmWeight() != this.getmWeight())
        return false;
    this.setPosition(pSceneTouchEvent.getX() - this.getWidth() / 2,
        pSceneTouchEvent.getY() - this.getHeight() / 2);
    if (pSceneTouchEvent.getAction() == TouchEvent.ACTION_UP) {
        checkForCollisionsWithTowers(this);
    }
    return true;
}
};

Ring ring2 = new Ring(2, 118, 212, this.mRing2, getVertexBufferObjectManager)
@Override
public boolean onAreaTouched(TouchEvent pSceneTouchEvent, float pTouchX, float pTouchY) {
    if (((Ring) this.getmStack().peek()).getmWeight() != this.getmWeight())
        return false;
    this.setPosition(pSceneTouchEvent.getX() - this.getWidth() / 2,
        pSceneTouchEvent.getY() - this.getHeight() / 2);
    if (pSceneTouchEvent.getAction() == TouchEvent.ACTION_UP) {
        checkForCollisionsWithTowers(this);
    }
    return true;
}
};

Ring ring3 = new Ring(3, 97, 255, this.mRing3, getVertexBufferObjectManager)
@Override
public boolean onAreaTouched(TouchEvent pSceneTouchEvent, float pTouchX, float pTouchY) {
    if (((Ring) this.getmStack().peek()).getmWeight() != this.getmWeight())
        return false;
    this.setPosition(pSceneTouchEvent.getX() - this.getWidth() / 2,
        pSceneTouchEvent.getY() - this.getHeight() / 2);
    if (pSceneTouchEvent.getAction() == TouchEvent.ACTION_UP) {
        checkForCollisionsWithTowers(this);
    }
    return true;
}
};
```

Notice that **onAreaTouched** returns a boolean value. When it returns true, the touch is consumed, otherwise it is passed down to other layers until someone consumes it. So the first thing you do in this method is check if the weight of the current ring is not equal to the weight of the first ring in the stack. If it is, that means this ring is the first element of the stack and so you can proceed to move it, otherwise you let the touch go since you can't move this ring.

You also check in **onAreaTouched** if the type of touch is an ACTION_UP event, triggered when the finger is lifted. If it is, you call **checkForCollisionsWithTowers** whose primary purpose is to check if the ring has collided with (or rather, is touching) a tower. As you may

recall, you added a placeholder for **checkForCollisionsWithTowers** already. Let's fill the function in now.

Replace the placeholder method for **checkForCollisionsWithTowers** with the following:

```
private void checkForCollisionsWithTowers(Ring ring) {
    Stack stack = null;
    Sprite tower = null;
    if (ring.collidesWith(mTower1) && (mStack1.size() == 0 ||
        ring.getmWeight() < ((Ring) mStack1.peek()).getmWeight())) {
        stack = mStack1;
        tower = mTower1;
    } else if (ring.collidesWith(mTower2) && (mStack2.size() == 0 ||
        ring.getmWeight() < ((Ring) mStack2.peek()).getmWeight())) {
        stack = mStack2;
        tower = mTower2;
    } else if (ring.collidesWith(mTower3) && (mStack3.size() == 0 ||
        ring.getmWeight() < ((Ring) mStack3.peek()).getmWeight())) {
        stack = mStack3;
        tower = mTower3;
    } else {
        stack = ring.getmStack();
        tower = ring.getmTower();
    }
    ring.getmStack().remove(ring);
    if (stack != null && tower != null && stack.size() == 0) {
        ring.setPosition(tower.getX() + tower.getWidth()/2 -
            ring.getWidth()/2, tower.getY() + tower.getHeight() -
            ring.getHeight());
    } else if (stack != null && tower != null && stack.size() > 0) {
        ring.setPosition(tower.getX() + tower.getWidth()/2 -
            ring.getWidth()/2, ((Ring) stack.peek()).getY() -
            ring.getHeight());
    }
    stack.add(ring);
    ring.setmStack(stack);
    ring.setmTower(tower);
}
```

The above code don't checks if the given ring is touching a tower, if it can be moved to that tower, or if the ring has been placed appropriately.

Build and run, and you should have a completely functional Tower of Hanoi game! There are only three rings, but that means you should be able to beat the game in no time. :P

The Source Code for this project:

<https://github.com/janebabra/TowerofHanoiProject>

libGDX

Setting up the project structure

Step 1:

Download the libgdx setup file:

<http://libgdx.badlogicgames.com/nightlies/dist/gdx-setup-ui.jar>

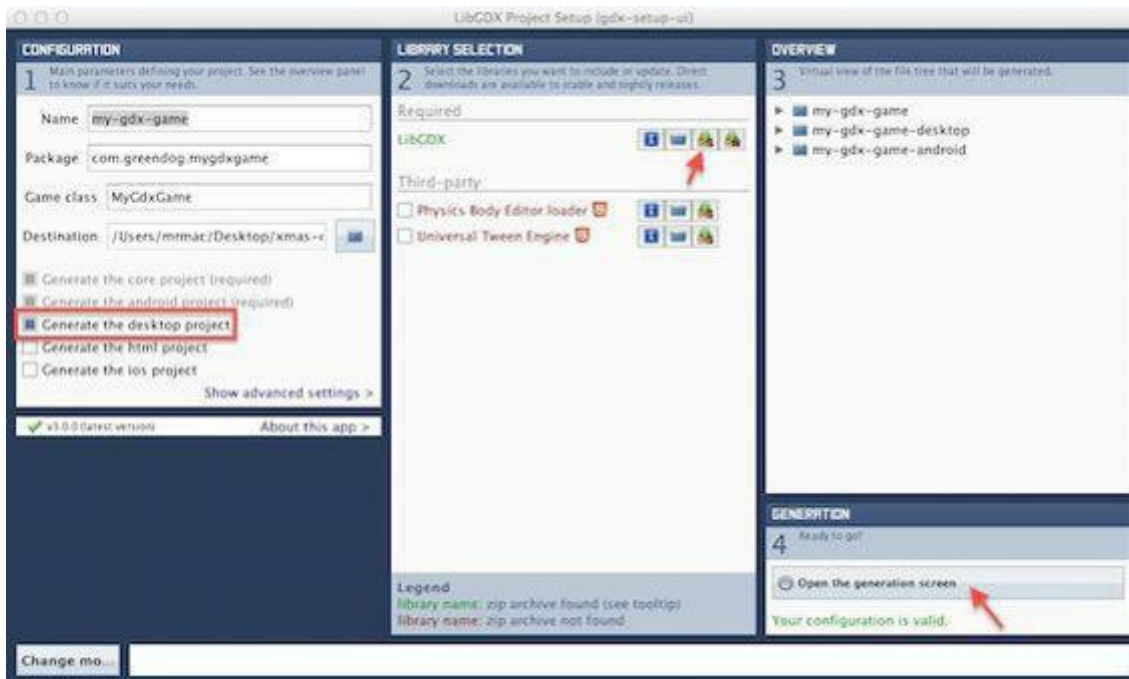
Step 2:

Create a project folder and copy the downloaded gdx-setup-ui.jar file into it.

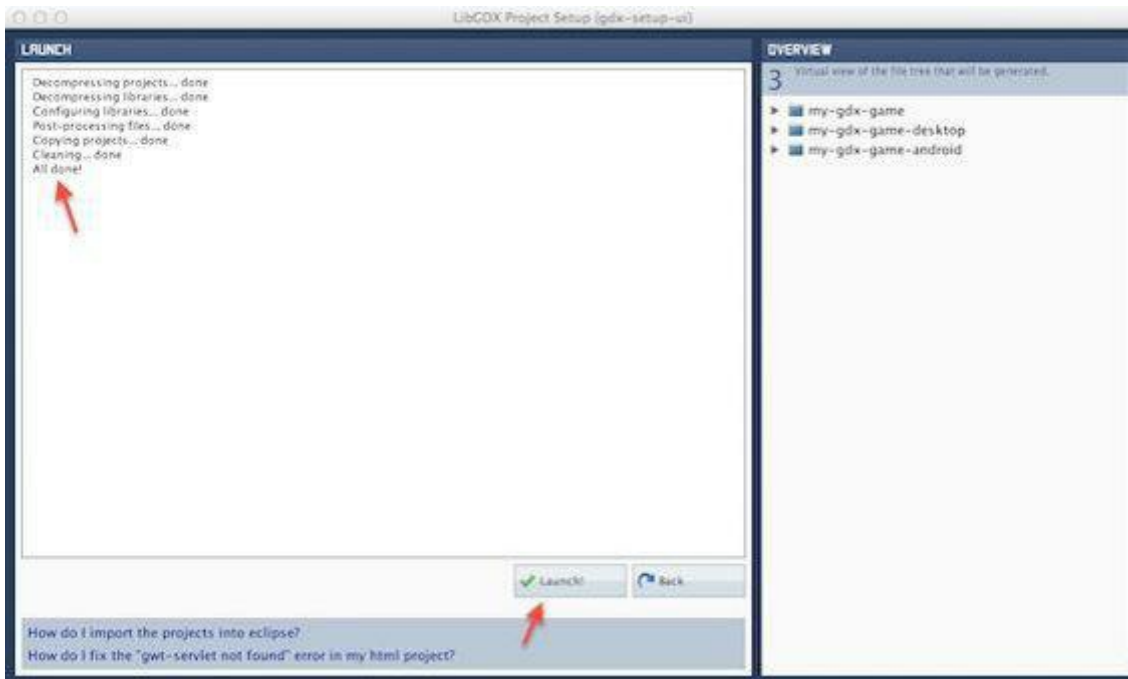
Then double click on it to launch it. If it does not work, try running `java -jar gdx-setup-ui.jar` from the command line.



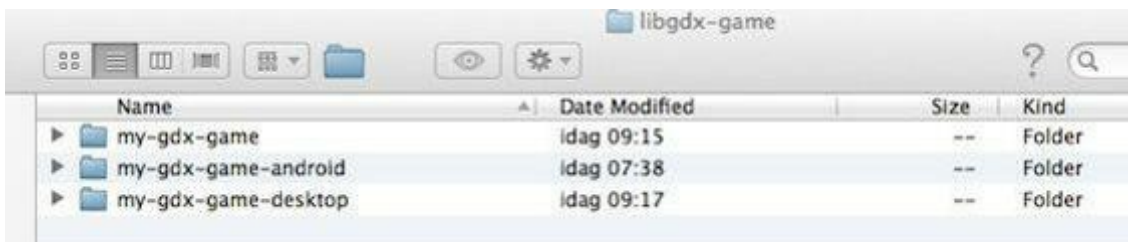
Click on Create.



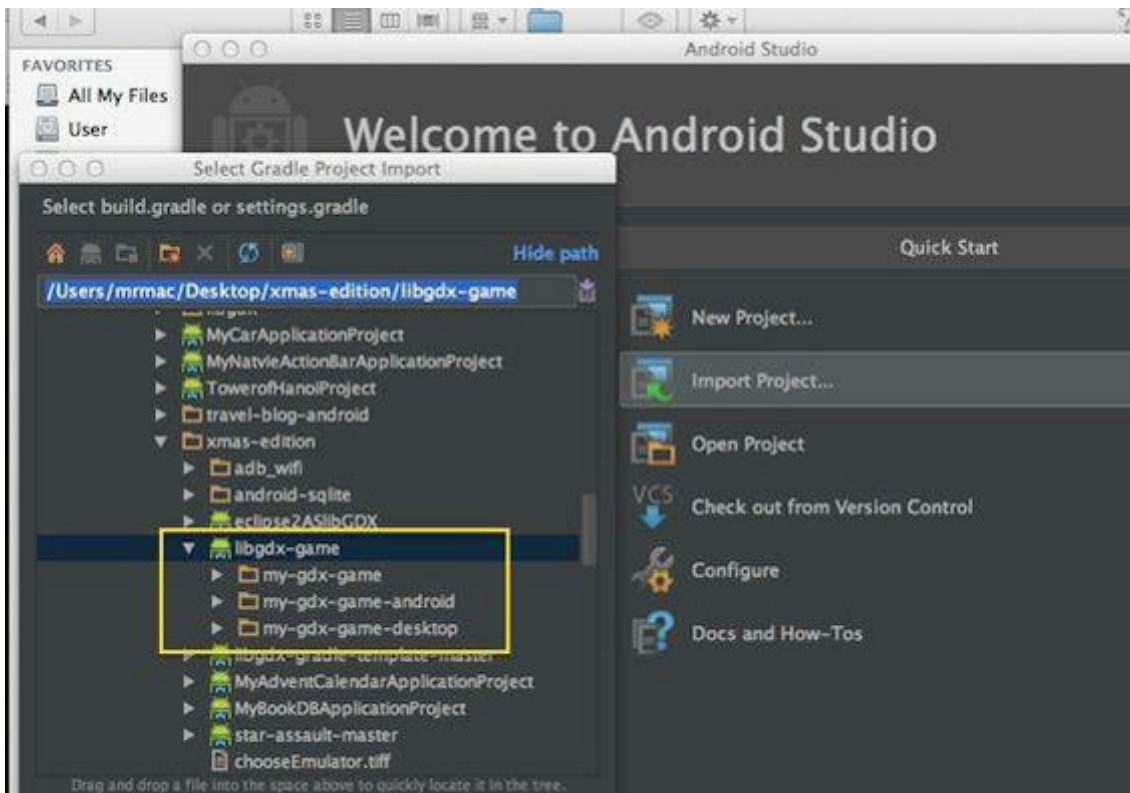
- From section 1 leave only the Generate the desktop project.
- In section 2 click on the marked right on LibGDX to download the libGDX library.
- Your section 3 should look similar to the screenshot.
- In section 4, click on “Open the generation screen”, than on Launch. This will build the project structure.



When you see “All done”, the setup is finished and the window can be closed.

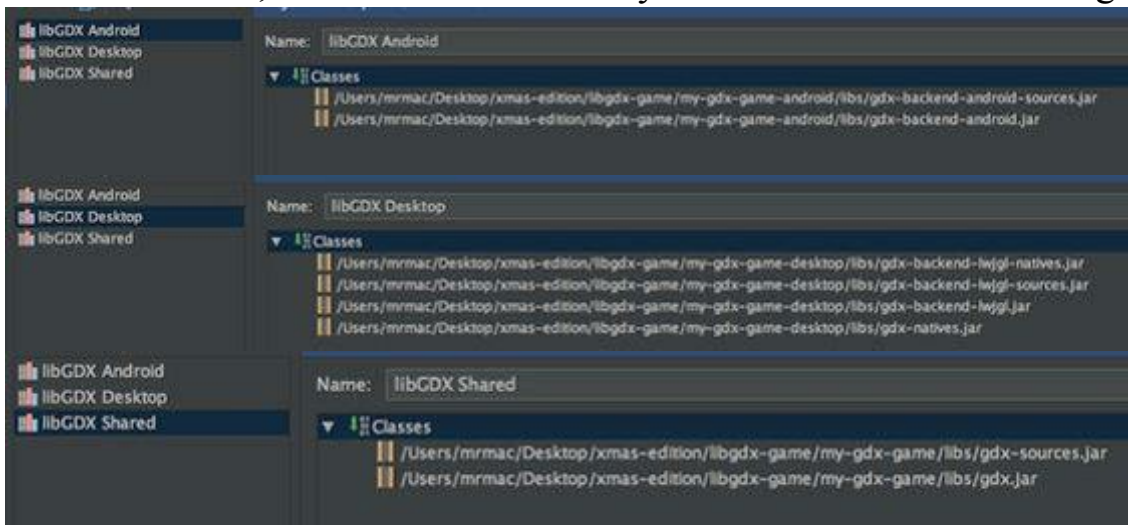


The project folder, how I called it, “libgdx-game” and the three folder which have been build by the setup. Now it is time to open up Android Studio and import the project. Import your earlier created libGDX-game-folder into Android Studio. From the Welcome Screen choose Import Project.



Take over all the suggestions and click Next until the Window Libraries – Library contents shows up.

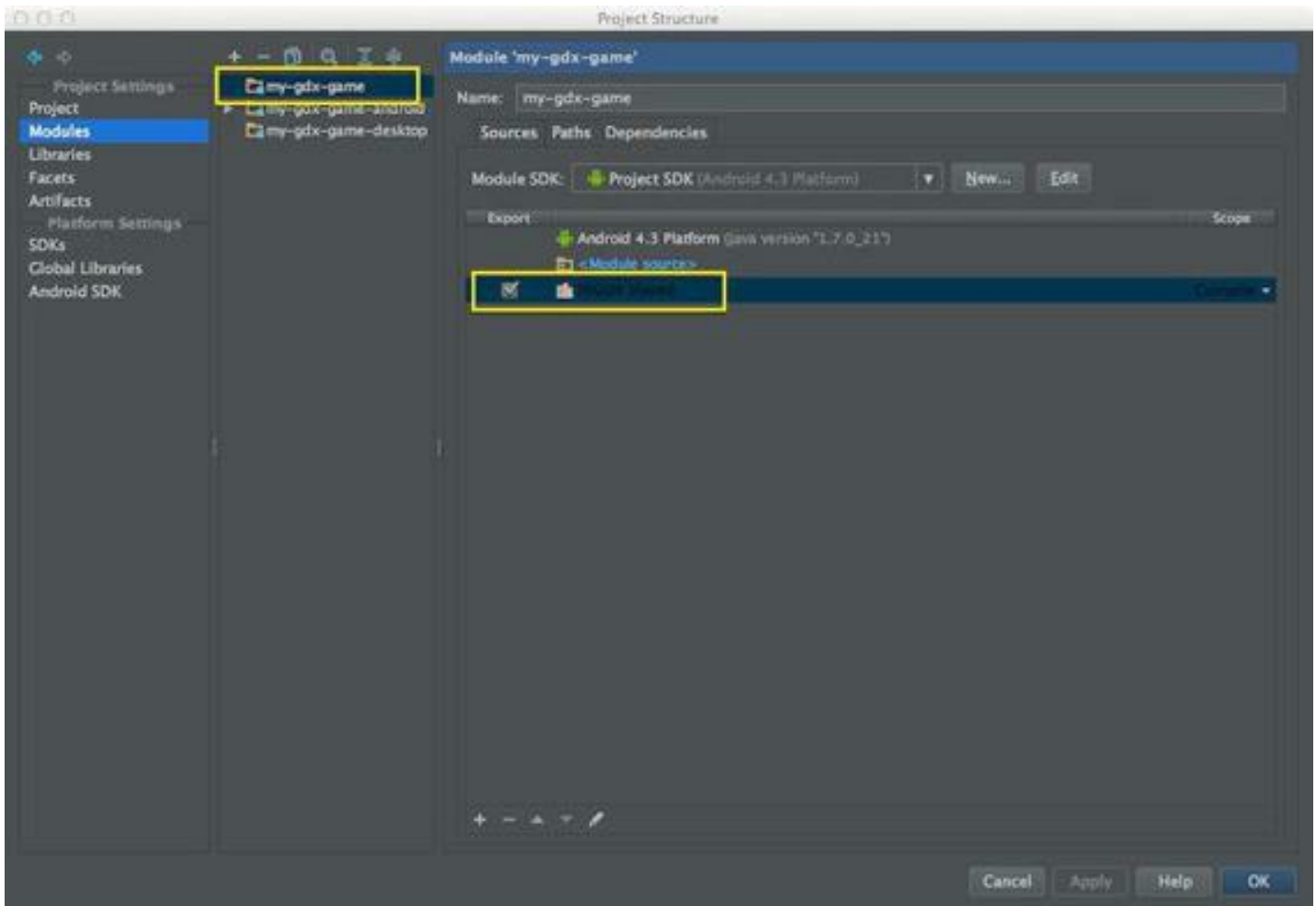
Rename the libs, so it matches the library contents like the 3 following screen shots shows.



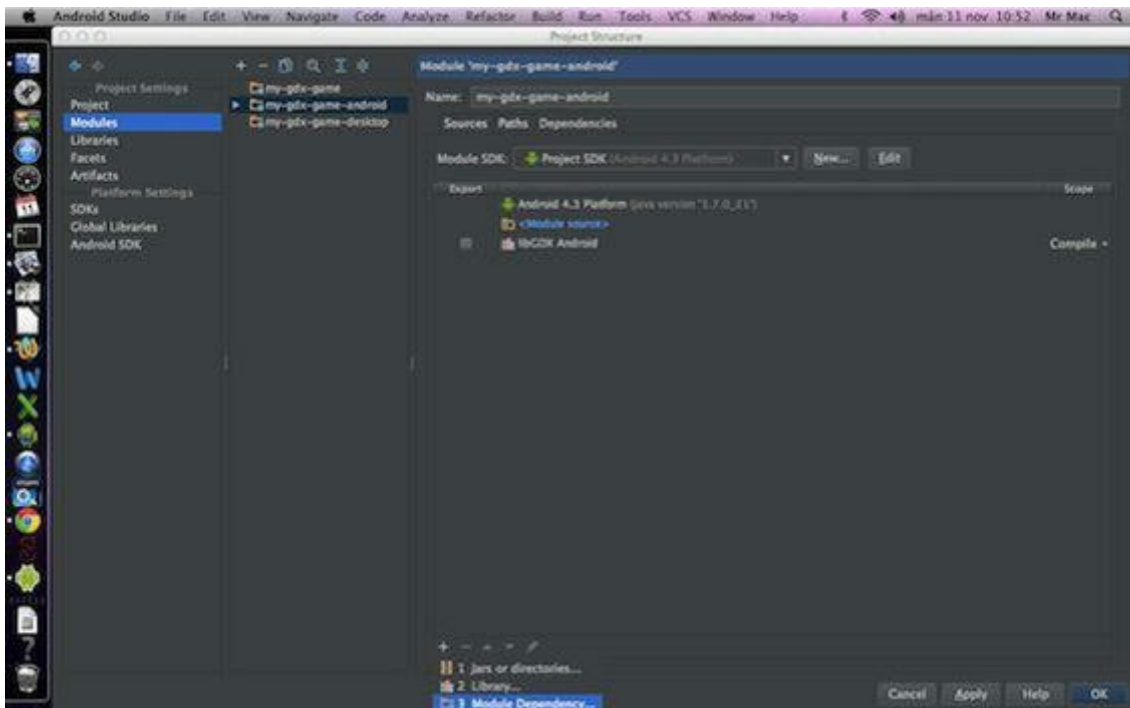
Continue with Next until Finish.

Modul settings

In the project structure panel right-click on the first folder and select Open Modul Settings.



Click next to libGDX Shared to export the library.



Go to the android module and add the main module as dependency by clicking on the + button on the bottom of the screen.



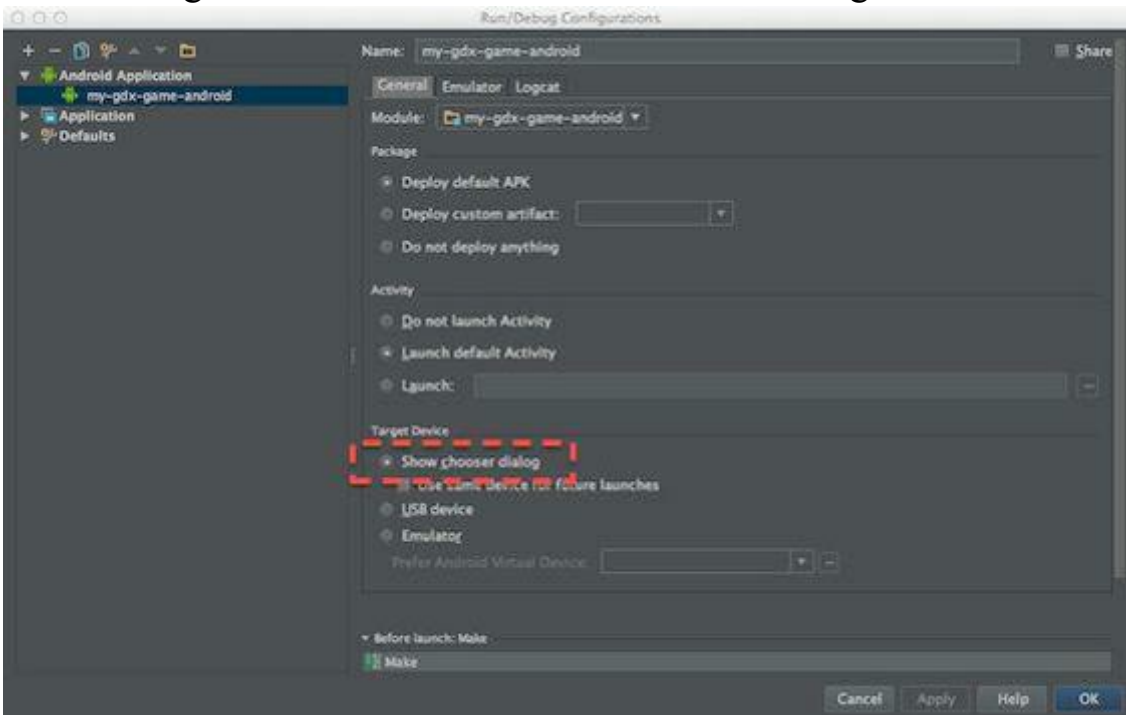
Select from the list the main module (my-gdx-game) and confirm with OK. The result should look like this.



Do the same for the desktop module and click OK.
Edit run configuration

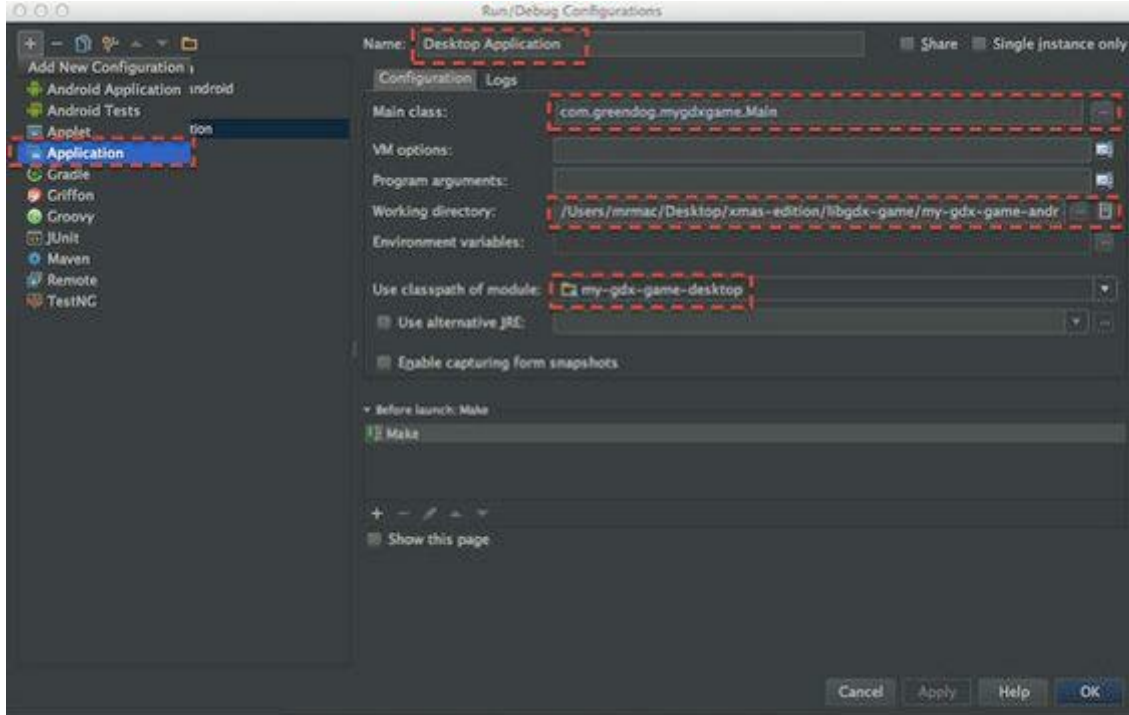
The run configuration for Android

Under “Target Device” activate “Show chooser dialog”

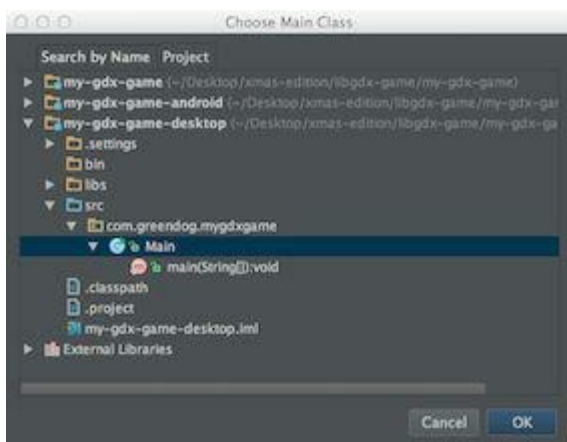


The Desktop run configuration

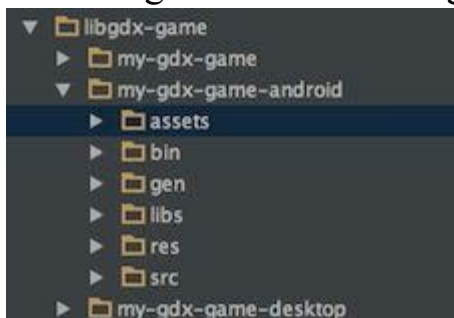
Click on the + button from the upper left corner of the screen and choose “Application” and name the new configuration.



Next go to Main class and choose from the dialog window the main class of the desktop starter. Click OK.



Now the working directory has to be changed. Click besides the Working directory field to get the dialog window and navigate to the android folder. Select the assets folder and click OK.



Last thing to do is to select the right classpath. Select here the desktop module and click OK.

Running the Desktop version



Running the Android version



You are good to go now.

This setup produces a working environment for developing, testing, running and building the Desktop and the Android version of a libGDX based game.

So, you can build APK's with this configuration.

But, you don't have any Gradle structure. If you need the Gradle features for some reason, you have to add the gradle files by hand.

There is a project on GitHub with a libGDX-gradle template. And the person who created it says, this will properly replace the gdx-setup-ui.jar we used in the beginning to setup the project structure. But unfortunately, in my test, I didn't get it to work.

Here is the Link from the project, so you can test it by yourself and follow the project.
<https://github.com/libgdx/libgdx-gradle-template>

The screenshot shows the small sample game from *badlogic* based on libGDX called Drop. The goal of the game is, to catch all the raindrops with the bucket.



The sample game Drop can be downloaded from GitHub:
<https://github.com/janebabra/drop>

The documentation for the game can be found here:
<https://github.com/libgdx/libgdx/wiki/A-simple-game>

Integration of JavaScript/ LIME games

The integration will be made by using WebView. WebView had been added to Android with API 1. So, integrated JavaScript games can be offered to older and newer phones as well.

First you need to add an assets folder to your folder structure.

The folder has to be under main/, on the same level like java/ and res/.

For the example, I used the sample game Roundball from the LIME Developer Site:

<http://www.limejs.com/>



By now, the whole roundball folder should be copied into the assets folder.

Next, the layout file needs a webview. This should be look like this:

```
<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"
    tools:context=".MainActivity" >
```

```
    <WebView
        android:id="@+id/webView1"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_alignParentLeft="true"
        android:layout_alignParentTop="true" />
```

```
</RelativeLayout>
```

Last but not least, the Java file. Games using the LIME GameEngine are using DOM instead of Canvas. Therefore, two of the webkit settings have to be added:

```
package com.greendog.androidstudio.example.roundballwebapp;
```

```
import android.os.Bundle;
```

```
import android.app.Activity;
import android.webkit.WebView;

public class MainActivity extends Activity {

    WebView myWebView;

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

        myWebView = (WebView) findViewById(R.id.webView1);
        myWebView.getSettings().setDomStorageEnabled(true);
        myWebView.getSettings().setJavaScriptEnabled(true);

        myWebView.loadUrl("file:///android_asset/roundball/roundball.html");

    }
}
```

No permission are needed when the files are local. You're ready to run the game!

The sample project can be found here:

<https://github.com/janebabra/Roundball>

Developing for Google TV

What is Google TV?

Google TV is a platform, a special customized Android Version for TVs. It has extensions you'll only need on TVs. like changing channels. But the rest of this platform is quite similar to Android versions for phone and tablets.

The Google TV Android version is not available for download. The Source Code is only send to OEM producer of Google TV hardware.

Google TV, often called smart TV comes in different shapes. Some TVs have Google TV already build-in, no extern device is needed. But in most cases, the Google TV is an extern device, a set-up box (buddy box) or HDMI-Stick, which are connected through HDMI to the TV.

When Google first presented Google TV in 2010, the intention was to give the end user a tool for streaming movies over the Internet without leaving the TV.

But if look how powerful this devices have become, you can imaging that this can replace a PC for a lot of people. A Google TV device with 2 GB Ram, up to 1,8 Ghz quad core CPU and 16 GB Flash Nand Memory for about \$ 60 is quite a catch.

The Software comes from the Play Store. Not every app is compatible with Google TV but in my tests, I could find apps, most people uses on their PC, like apps for:

- Movies
- Pictures
- Music
- Internet browsing
- Email
- Calendar
- Facebook
- Twitter
- Google+
- Office
- Google Drive
- Dropbox
- Ebook reading
- Games
- and more

What does it mean for developers? I think, it gives developers a broader selection of apps to develop.

A closer look at the Google TV hardware.

Most devices have Micro SD Slots that can take up to 32 GB. There have minimum 1 x USB and up to 4. The USB can be used for connecting wireless keyboard/ mouse, USB memory sticks and cable to connect to the computer. Only few Google TV devices are able to connect to a webcam.

Setup for the development

General information's from Google can be found at the following Webpage:

Google TV developers webpage

<https://developers.google.com/tv/>

To develop Android apps for Google TV your development IDE needs to have the Android SDK Platform Tools rev. 6 or higher and the Android SDK Tools rev. 12 or later are needed. Android Studio gives both.

There's an Add-on called "Google TV Add-on". This Add-on is not needed anymore for developing apps. But in some cases it can be useful for the Emulator. The Google TV Add-on extends the Emulator with extra TV-navigation-Buttons. When you for example develop games without any TV features, the Google TV Add-on is not necessary at all.

Google TV Add-On

https://developers.google.com/tv/android/docs/gtv_addon

If you need the Google TV Add-on for your Emulator, install it from the SDK Manager. There're only 2 versions, one for Android API 12 and one for Android API 13. To be able to create an Emulator with Google TV Add-on, you'll need the rest of the API too.

The Project minSDK should be API 12 or higher.

As targetSDK is recommended API 13 or higher.

Google TV examples

1. Google TV Remote
<https://code.google.com/p/google-tv-remote/>
2. Google TV sample code
<https://code.google.com/p/google-tv-android-samples/>

The examples are very old and a little bit outdated and all of them are Eclipse projects.

The Google TV Remote example is quite interesting because not every Google TV device has its own remote. People can either buy a remote or install the Google TV Remote app on their Android phone. But unfortunately none of my Android phone could download and install the Remote from Google Play. I'm sure; I'm not the only one who can't download the old Remote.



Here's the Link to the Google Play Store:

<https://play.google.com/store/apps/details?id=com.google.android.apps.tvremote>

So, here's what I did to convert the old project into a new Android Studio project:
(To load the project direct in Android Studio 0.4 didn't work)

1. Import into Eclipse
2. Export from Eclipse with Gradle
3. Import into Android Studio
4. The export from Eclipse and import into Android Studio didn't work to 100%. The **library protobuf-java-2.2.0-lite.jar** hasn't be copied into libs/. I had to copy this file

manually into libs/.

5. And a small correction in the *build.gradle* was necessary:

```
dependencies {  
    compile files('libs/protobuf-java-2.2.0-lite.jar')  
    compile files('libs/anymote.jar')  
    compile files('libs/bcprov-jdk15-143.jar')  
    compile files('libs/polo.jar')
```

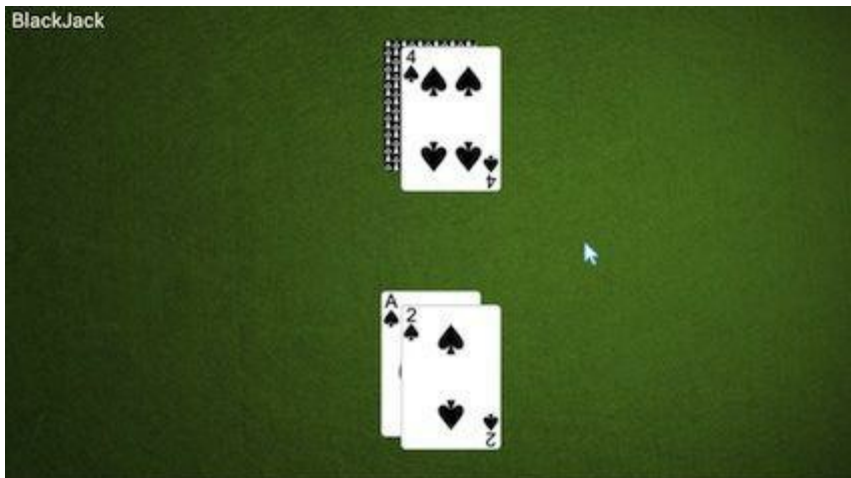
The Source Code for Google's TV remote as Android Studio project can be found on my GitHub:

<https://github.com/janebabra/gtv/tree/master/google-tv-remote>

This strategy will properly work for other examples too.

The *LeftNavBarLibrary* from the examples should be replaced by Fragments.

The BlackJack example is quite interesting. It's an example for the "second screen". The project **BlackJackGTV** is the BlackJack Table showing on the TV. The Eclipse project BlackJackGTV can be imported directly into Android Studio 0.4 or higher.



The project **BlackJackTVRemote** is the Android Phone App. The game play is on the phone and the result of any action is showing on the TV screen. The project BlackJackTVRemote depends on the project AnymoteLibrary. This means, when using Android Studio, the AnymoteLibrary has to be imported into the BlackJackTVRemote project, similar to ActionBarSherlock.



How to test an app running on TV

Like with any other Android device the **adb** needs to connect to the device either via USB or via WiFi (tcpip). If your Google TV device has a free USB you can connect it via USB cable with your computer. But because the TV is often not right in front of your computer, you may prefer the WiFi connection.

Setting up the adb WiFi connection

The default for adb is the USB connection. To change the connectivity type to WiFi, you need

- connect your phone via USB with the computer
- open a terminal window and navigate into the platform-tools folder
- with dir (Windows) or ls (Linux/ Mac) you should see the adb
- still in a terminal window, use the command
- **adb tcpip 5555** or **./adb tcpip 5555 (Mac, Linux)**

```
mr-macs-macbook:platform-tools mrmac$ ls
NOTICE.txt      api              source.properties
adb             fastboot        systrace
mr-macs-macbook:platform-tools mrmac$ ./adb tcpip 5555
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
restarting in TCP mode port: 5555
□
```

now in your Google TV

go to Settings -> About -> Status to find the IP-Address

for example, the IP-Address is 192.168.0.103

in the Terminal window use the command

adb connect 192.168.0.103

```
mr-macs-macbook:platform-tools mrmac$ ./adb connect 192.168.0.103
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
connected to 192.168.0.103:5555
□
```

From now on, you can run the app and see the Google TV device in *Choose Device*:



Run the App, monitor and make screenshots.

```
Android Studio
Logcat
Log level: Verbose
18-23 03:36:13.676 2822-2862/? S/W/Sensor: set_power_states = M$StoreCalibration()
18-23 03:36:13.676 2822-2862/? S/W/Sensor: [void set_power_states(int) @ 0: 0]
18-23 03:36:13.676 2822-2862/? S/W/Sensor: [void set_M$Sensor::enableCalibK_Fc(int) @ 0: 0]
18-23 03:36:13.676 2822-2862/? S/SensorService: SensorDevice::activate (enable 1, disable 0) = 0
18-23 03:36:13.676 2822-2862/? S/SensorService: activating sensor disable handler4
18-23 03:36:13.676 2822-2862/? S/SensorService: activating sensor disable handler
18-23 03:36:13.676 2822-2862/? S/SensorService: SensorDevice::activating sensor handler4 op=2888888888
18-23 03:36:13.676 2822-2862/? S/W/Sensor: setDelay / handler4, delay=2888888888
18-23 03:36:13.676 2822-2862/? S/: setLight_backlight is called!!
```


Things to think about when developing from Scratch

The TV screen is much bigger in size than a phone or tablet but not in the resolution. The screen resolution is either 720p or 1080p.

720p means:

- 1280 x 720 px screen size in pixel
- **tvdpi** is the qualifier for the image resource folder, for example drawable-tvdpi
- the screen density is 213 dp
- the qualifier for the layout folder is **large**, like layout-large

1080p means:

- 1920 x 1080 px screen size in pixel
- **xhdpi** is the qualifier for the image resource folder
- the screen density is 320 dp
- the qualifier for the layout folder is **large**

The TV is always in landscape.

And because People are sitting some feet away from the TV, the layout must not be overloaded. Big text, big buttons. The navigation should be possible with a D-Pad (up, down, left, right, OK). Not everybody has a mouse.

Google make a lot of recommendations for the layout, but unfortunately their don't follow these recommendations with their own apps.

For example "Light text on a dark background is slightly easier to read on a TV". But when you look at the Google apps like the Browser or Gmail, you have white background and black text.

Especially the colors are looking different on TV. Therefore it's important to test the app on TV and not only on the Emulator.

Of course, the Emulator can be helpful too. Best choice for testing on an Emulator is the device

"10.1 WXGA (Tablet) (1280 x 800: mdpi) "

Android Code Templates

What are Android Code Templates?

An Android Code Template generates a boilerplate Android Code and assets from the input of simple UI parameters.

The Templates are written using **FreeMarker**, a Java templating engine.

The default ADT templates integrated in Android Studio are divided into 3 different types:

1. Android Application Templates
2. Android Activity Templates
3. Android Object Templates

Android Application Templates:

can be reached when via New Project and offer the 2 choices, Android Application and Android Library.

Android Activity Templates:

from the Project Wizard:

- Blank Activity
- Full Screen Activity
- Master/ Detail Flow
- your own customized templates, like the None-Fragment Template we'll make later.

from the context menu New -> Activity (right-click on the package):

- Blank Activity
- Full Screen Activity
- Login Activity
- Master/ Detail Flow
- Settings

Android Object Templates:

from the context menu New -> Android Component (right-click on the package):

- New App Widget
- New Blank Fragment
- Broadcast Receiver
- Content Provider
- Custom View
- New Daydream
- New Intent Service
- New ListFragment
- Service

Where are the ADT Templates located?

The Templates are under your Android Studio installation folder, in plugins, android, lib, templates.

Under templates are the folders activities, projects and others, where the different types of templates are stored.

- projects: Android Application Templates
- activities: Android Activity Templates
- others: Android Object Templates

Making Custom Code Templates

For Eclipse there exist a FreeMarker IDE plugin, but not yet for Android Studio.

In the meantime, you can create new templates with your favorite text editor.

The following example for creating a None-Fragment Activity Template shows an easy way to do this:

The simplest way to create a new template is to make a copy of `BlankActivity` and rename it to whatever you like - in our case *NoneFragmentActivity*.

This will give you a few icons and other bits and pieces that you don't really need in your finished template, but it is much easier than starting from scratch.

At the End you can choose the New Template from the Project Wizard:



Template.xml

Just adding the *NoneFragmentActivity* folder will cause Android Studio to list Blank Activity twice in the Activity types. To give your template an identity of its own, the first file you have to edit is *Template.xml*.

If you load this into Android Studio or any XML editor you should be able to figure out what it is all about. It controls the options that the user is presented with as the final page of the New Project screen.

In our case all we need to modify is the name variable to give the template a new name and we need to remove a number of parameters that are no longer needed.

We need to remove the parameter `navType`, and the parameter `fragmentLayoutName`. Everything else can be left as it is, but there are some optional changes you might want to make to tidy things up.

You can find details of what sorts of parameters etc. you can define by looking at the documentation. Once you have seen an example it is fairly obvious.

The final *Template.xml* is:

```
<?xml version="1.0"?>
<template
    format="3"
    revision="1"
    name="Simple Blank None Fragment Activity"
    minApi="7"
    minBuildApi="14"
    description="Creates a new simple blank none-fragment activity">
<category value="Activities" />
<parameter
    id="activityClass"
    name="Activity Name"
    type="string"
    constraints="class|unique|nonempty"
    suggest="`${layoutToActivity(layoutName)}"`
    default="MainActivity"
    help="The name of the activity class to create" />
<parameter
    id="layoutName"
    name="Layout Name"
    type="string"
    constraints="layout|unique|nonempty"
    suggest="`${activityToLayout(activityClass)}"`
```

```

        default="activity_main"

        help="The name of the layout to create for the activity" />
<parameter

    id="activityTitle"

    name="Title"

    type="string"

    constraints="nonempty"

    default="MainActivity"

    suggest="{activityClass}"

        help="The name of the activity. For launcher activities, the
application title." />
<parameter

    id="isLauncher"

    name="Launcher Activity"

    type="boolean"

    default="false"

        help="If true, this activity will have a CATEGORY_LAUNCHER intent
filter, making it visible in the launcher" />
<parameter

    id="parentActivityClass"

    name="Hierarchical Parent"

    type="string"

    constraints="activity|exists|empty"

    default=""

        help="The hierarchical parent activity, used to provide a default
implementation for the 'Up' button" />
<parameter

```

```

    id="packageName"

    name="Package name"

    type="string"

    constraints="package"

    default="com.mycompany.myapp" />

    <!-- 128x128 thumbnails relative to template.xml -->

    <thumb>template_blank_activity.png</thumb>

    <globals file="globals.xml.ftl" />

    <execute file="recipe.xml.ftl" />

</template>

```

The *globals.xml.ftl* file sets up some variables which apply to all of the template files. In general you don't have to modify this as any global variables that it defines you can either use or ignore.

What does matter however is the *recipe.xml.ftl* file, which is the script that sets up the project.

Recipe.xml.ftl

The Recipe file does all of the work setting up the project, but it does most of this by copying and expanding directories which exist in the template root directory.

One key command that you are likely to use is `copy`, which copies a directory and all its subdirectories from the template root to the project. The command `instantiate` does the same job as `copy`, but it expands any FreeMarker tags within the files. The final command is `merge`, which as you can guess merges the content of a source file in the template root with a project file.

The Recipe file starts out by loading compatibility libraries if they are needed:

```

<?xml version="1.0"?>

<recipe>

    <#if appCompat?has_content>

        <dependency

```



```

        mavenUrl="com.android.support:appcompat-v7:+"/> </#if>
<#if !appCompat?has_content &&
            hasViewPager?has_content>
    <dependency
        mavenUrl="com.android.support:support-v13:+"/>
</#if>
<#if !appCompat?has_content >
    <dependency
        mavenUrl="com.android.support:support-v4:+"/>
</#if>

```

You might as well leave these lines in your own Recipe file.

Next we need to create the manifest:

```

<merge from="AndroidManifest.xml.ftl"
    to="{escapeXmlAttribute(manifestOut)}/
        AndroidManifest.xml"
/>

```

The template manifest is stored in the root:

```

<manifest xmlns:android=
    "http://schemas.android.com/apk/res/android" >
<application>
    <activity android:name=
        "${packageName}.${activityClass}"
    <#if isNewProject>
        android:label="@string/app_name"
    <#else>

```

```

        android:label="@string/title_${
            activityToLayout(activityClass)}"
    </#if>
    <#if buildApi gte 16 &&
        parentActivityClass != "">
        android:parentActivityName=
            "${parentActivityClass}"
    </#if>
>
<#if parentActivityClass != "">
    <meta-data android:name=
        "android.support.PARENT_ACTIVITY"
        android:value="${parentActivityClass}"
    />
</#if>
<#if isLauncher>
    <intent-filter>
        <action android:name=
            "android.intent.action.MAIN" />
        <category android:name=
            "android.intent.category.LAUNCHER" />
    </intent-filter>
</#if>
</activity>
</application>
</manifest>

```

When this file is processed it is merged with any existing manifest file in such a way that only the new lines are added. What this means is that when the same template is used to create a new activity within an existing project only the new definition of the activity is added to the existing manifest.

Next we have to merge some resource files:

```
<merge from="res/values/strings.xml.ftl"
      to="{escapeXmlAttribute(resOut)}/
          values/strings.xml" />
<merge from="res/values/dimens.xml.ftl"
      to="{escapeXmlAttribute(resOut)}/
          values/dimens.xml" />
<merge from="res/values-w820dp/dimens.xml"
      to="{escapeXmlAttribute(resOut)}/
          values-w820dp/dimens.xml" />
```

As before, the values are merged with any existing files so there is no duplication of definitions.

The strings.xml file is where we put the "Hello world" message:

```
<resources>
  <#if !isNewProject>
    <string name=
      "title_{{activityToLayout(activityClass)}}">
      {{escapeXmlString(activityTitle)}}</string>
    </#if>
  <string name="hello_world">Hello world!</string>
</resources>
```

The dimens file just contains some basic layout properties;

```
<resources>
```

```

<!-- Default screen margins,
     per the Android Design guidelines. -->
<dimen name=
     "activity_horizontal_margin">16dp</dimen>
<dimen name=
     "activity_vertical_margin">16dp</dimen>
</resources>

```

If you want to define additional string resources or layout parameters simply add them to the appropriate file.

Our final task is to create the Java source and the xml layout files and this is just a matter of copying and expanding two template files:

```

<instantiate from=
     "res/layout/simpleactivity.xml.ftl"
to="{escapeXmlAttribute(resOut)}/
     layout/{layoutName}.xml" />
<instantiate from=
     "src/app_package/SimpleActivity.java.ftl"
to="{escapeXmlAttribute(srcOut)}/
     {activityClass}.java" />

```

The layout file template *simpleactivity.xml* (Hello World) is very simple:

```

<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:paddingLeft=

```

```

        "@dimen/activity_horizontal_margin"
    android:paddingRight=
        "@dimen/activity_horizontal_margin"
    android:paddingTop=
        "@dimen/activity_vertical_margin"
    android:paddingBottom=
        "@dimen/activity_vertical_margin"
    tools:context="${packageName}.${activityClass}"><TextView
    android:text="@string/hello_world"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
</RelativeLayout>

```

You can modify `RelativeLayout` to be any sort of container you want and you can remove the "hello world" message if you find it gets in the way.

The Java template is:

```

package ${packageName};

import

<#if appCompat?has_content>
    android.support.v7.app.ActionBarActivity
<#else>android.app.Activity
</#if>;

import android.

<#if appCompat?has_content>support.v7.</#if>
app.ActionBar;

import android.os.Bundle;

import android.view.LayoutInflater;

```

```

import android.view.Menu;

import android.view.MenuItem;

import android.view.View;

import android.view.ViewGroup;

import android.os.Build;public class ${activityClass} extends Activity {

    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.${layoutName});

    }

}

```

As before, you can modify this to include features you might want, but this the very minimum.

Finally if you want to open some files in the editor ready for the programmer to start work you can:

```

<open file=

    "${escapeXmlAttribute(srcOut)}/

        ${activityClass}.java" />

</recipe>

```

This opens the java souce code ready to work.

And this is the end of the recipe file and the entire template.

The sample can be downloaded from GitHub:

<https://github.com/janebabra/NoneFragmentActivity>

Template Documentation

The **FreeMarker** documentation can be found here:

<http://freemarker.org/docs/index.html>

Google's Template Documentation

I cloned the side from Google:

<http://google-android-studio.blogspot.se/p/android-code-templates.html>

The original side:

<https://android.googlesource.com/platform/sdk/+/refs/heads/master/templates/docs/index.html>

Help and further information

Inside of Android Studio

From the Help menu and from the „Tip of the Day“.

Outside from the Android Studio:

Android Tools Project Site

Release Status

<http://tools.android.com/overview>

AVD- Emulator

<http://developer.android.com/tools/devices/emulator.html>

<http://developer.android.com/tools/devices/emulator.html#acceleration>

Android Developers Blog

<http://android-developers.blogspot.se/2013/05/android-studio-ide-built-for-android.html>

Welcome to Google Developers Live

<https://developers.google.com/live/>

The App Example for Android Studio from the I/O Google

<https://github.com/bradabrams/stopwatchio13>

Explains the Example

<http://bradabrams.com/2013/06/google-io-2013-demo-android-studio-cloud-endpoints-synchronized-stopwatch-demo/>

Ultimate Stopwatch Apps (Eclipse)

<https://code.google.com/p/android-ultimatestopwatch/>

Overview of Google Cloud Endpoints

<https://developers.google.com/appengine/docs/java/endpoints/>

Google Cloud Console

<https://cloud.google.com/console>

From the **Google API console**, click Overview and locate the project number for your project:

<https://code.google.com/apis/console>

Google Cloud Messaging for Android Docu

<http://developer.android.com/google/gcm/index.html>

Google+ Community Android Studio

<https://plus.google.com/u/0/communities/101262515781041757195>

Issue Tracker

<https://code.google.com/p/android/issues/list>

Google's GIT Repository for Android Studio

<https://android.googlesource.com/platform/tools/adt/idea/>

IntelliJ Webhelp

<http://www.jetbrains.com/idea/webhelp/getting-help.html>

Google Analytics for Android Apps

<https://developers.google.com/analytics/devguides/collection/android/resources>

Flurry Analytics for Android Apps

<http://www.flurry.com/flurry-analytics.html>

Gradle Plugin User Guide

<http://tools.android.com/tech-docs/new-build-system/user-guide>

User Guide from gradle.org

<http://www.gradle.org/docs/current/userguide/userguide.html>

Multiple APK Support

<http://developer.android.com/google/play/publishing/multiple-apks.html>

Build Server and Test Server

- AppThwack
<https://appthwack.com/>
- TestDroid
<http://testdroid.com/>
- InstrumentTest example
<https://www.manymo.com/pages/blog/android-testing-in-the-cloud>
- Manymo
<https://www.manymo.com>
- Travis
<https://travis-ci.org/>