**Android Development Code Guidelines**

Naming conventions for Method

Camel case

Button Identifier: - btn\_observation\_add (tool\_entity\_action)

TextView

ImageView

CardView

RecyclerView

ListView

Id in XML

Useful links: Must read for Investickations, Wapp, Slate, Revival and Fbite

<http://developer.android.com/guide/components/index.html>

<https://developer.android.com/sdk/installing/migrate.html>

<http://developer.android.com/guide/appendix/media-formats.html>

<http://developer.android.com/guide/topics/ui/menus.html>

<http://developer.android.com/guide/components/fragments.html#Adding>

<http://developer.android.com/guide/topics/manifest/manifest-intro.html>

Performance tools

<http://developer.android.com/tools/performance/index.html>

Testing

<http://developer.android.com/tools/testing/testing-tools.html>

Ram usage

<http://developer.android.com/tools/debugging/debugging-memory.html>

<http://developer.android.com/tools/performance/comparison.html>

<http://developer.android.com/tools/performance/heap-viewer/index.html>

<http://developer.android.com/tools/performance/allocation-tracker/index.html>

Support Library

<http://developer.android.com/tools/support-library/index.html>

Fragments

<http://developer.android.com/guide/components/fragments.html>

Processes and Threads

<http://developer.android.com/guide/components/processes-and-threads.html>

Location and Sensors

<http://developer.android.com/guide/topics/sensors/index.html>

Building for multiple screens

<http://developer.android.com/training/best-ui.html>

Graphics and Animations

<http://developer.android.com/training/building-graphics.html>

Connectivity and Cloud

<http://developer.android.com/training/building-connectivity.html>

It's used when the Activity is forcefully terminated by the OS (ex: when your Activity is in the background and another task needs resources). When this happens, onSaveInstanceState(Bundle outstate) will be called and it's up to your app to add any state data you want to save in outstate.

When the user resumes your Activity, onCreate(Bundle savedInstanceState) gets called and savedInstanceState will be non-null if your Activity was terminated in a scenario described above. Your app can then grab the data from savedInstanceState and regenerate your Activity's state to how it was when the user last saw it.

Basically in onCreate, when savedInstanceState is null, then it means this is a 'fresh' launch of your Activity. And when it's non-null (if your app saved the data in onSaveInstanceState(...), it means the Activity state needs to be recreated.

import android.support.v4.app.Fragment;

This will call method getSupportFragmentManager()

getSupportFragmentManager().beginTransaction().add(R.id.guide\_fragment\_container, guideIndexFragment).commit();

while

import android.app.Fragment;

will call method getFragmentManager()

getFragmentManager().beginTransaction().add(R.id.guide\_fragment\_container, guideIndexFragment).commit();

getSupportFragmentManager() is not part of Fragment, so you cannot get it here that way. You can get it from parent Activity (so in onAttach() the earliest) using normal

activity.getSupportFragmentManager();

Use getActivity().getSupportFragmentManager()

Android fragment example

What is an Android Fragment?

Well as the name suggests it just a piece of something and in case of android its a piece of an Activity. Just like an activity the fragment also has a very similar life cycle onCreate(), onStart(), onStop(), onDestroy(), etc. In addition to that it has onCreateView() and onActivityCreated(). The system calls the onCreateView() when it's time for the fragment to draw its user interface for the first time. In this method you must inflate your fragment view return it. You can return null if the fragment does not provide a UI. The onActivityCreated() method is called when the fragment's activity has been created.

Why would we use a fragment?

You can create all your logic in an Activity and then manipulate the screen based on device size and orientation but the amount of code to programmatically draw views and implement all the logic based on layout and/or orientation will be make the whole project more complex than it needs to be. This is where fragments come to rescue, you can separate out the logic based on a functional entity rather than how the whole application looks to the end user. The concept is similar to the Portal technology where you create more than one portlet and put them on a given page.

The example

we are going to implement how to display a list view and a detail view displayed side by side when the user is in a landscape mode and in the case of portrait mode we just display the list and then when the user clicks on the given entry we display the detail view. The list entries in this tutorial are web URLs and actual web page is the detail view. The objective is to create the two fragments that are totaly independent and don't talk to each other which was done with the help of implementing an interface on the list activity. The activity is notified when the user select an URL and it alerts the detail fragment to updates its view. In addition to that we dynamically add and remove fragments from the activity based on the layout and the user action. So in the case of the portrait mode we only inflate the list view and then swap it out when the user clicks on the URL link whereas in the landscape mode we inflate both views the frame layouts. This example doesn't use another activity to display the detail page as some other tutorials have used, the objective is to have a simple and easy to understand tutorial that covers all the dynamics of the power of Android Fragments and how to use them effectively without complicating the matters.

==============================================================================================================

==============================================================================================================

apply plugin: 'com.android.application'

android {

compileSdkVersion 21

buildToolsVersion "21.0.1"

defaultConfig {

applicationId "master.com.sfsu.investickation"

minSdkVersion 19

targetSdkVersion 21

versionCode 1

versionName "1.0"

}

buildTypes {

release {

minifyEnabled false

proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'

}

}

}

dependencies {

compile fileTree(dir: 'libs', include: ['\*.jar'])

compile 'com.android.support:support-v4:22.1.1'

compile 'com.google.android.gms:play-services:7.0.0'

compile "com.android.support:appcompat-v7:21.0.+"

compile 'com.android.support:cardview-v7:21.0.+'

compile 'com.android.support:recyclerview-v7:21.0.+'

compile 'com.jakewharton:butterknife:6.1.0'

}

AppCompat v21 builds themes that require the new APIs provided in API 21 (Android 5.0). To compile your application with AppCompat, you must also compile against API 21. The recommended setup for compiling/building with API 21 is a compileSdkVersion of 21 and a buildToolsVersion of 21.0.1 (which is the highest at this time - you always want to use the latest build tools).

Make sure the value for target (which tells the target android version) in project.properties file of both your project folder and appcompat\_v7 folder is same (preferably the latest).

: inside 'your\_project'/project.properties

target=android-21

android.library.reference.1=../appcompat\_v7

and

: inside appcompat\_v7/project.properties

target=android-21

android.library=true

and after this don't forget to clean your project .

Changing compile 'com.android.support:appcompat-v7:21.0.0' into compile 'com.android.support:appcompat-v7:20.0.0' in gradle.build works for me.

==============================================================================================================

==============================================================================================================

<PROJECT\_ROOT>\app\build.gradle is specific for app module.

<PROJECT\_ROOT>\build.gradle is a "Top-level build file" where you can add configuration options common to all sub-projects/modules.

If you use another module in your project, as a local library you would have another build.gradle file: <PROJECT\_ROOT>\module\build.gradle

For example you can use your top level file to specify these common properties:

buildscript {

repositories {

mavenCentral()

}

dependencies {

classpath 'com.android.tools.build:gradle:0.9.+'

}

}

ext {

compileSdkVersion = 19

buildToolsVersion = "19.0.3"

}

In your app\build.gradle

apply plugin: 'android'

repositories {

mavenCentral()

}

android {

compileSdkVersion rootProject.ext.compileSdkVersion

buildToolsVersion rootProject.ext.buildToolsVersion

}

==============================================================================================================

==============================================================================================================

...

Project Build File

<PROJECT\_ROOT>\build.gradle or the Project Build File is for the entire project so it will be used for global project configurations. A typical Project Build File contains the following:

buildscript which defines:

repositories and

dependencies

Gradle Plugin version

By default, the project-level Gradle file uses buildscript to define the Gradle repositories and dependencies. This allows different projects to use different Gradle versions. Supported repositories include JCenter, Maven Central, or Ivy. This example declares that the build script uses the JCenter repository and a classpath dependency artifact that contains the Android plugin for Gradle version 1.0.1.

Module Build File

<PROJECT\_ROOT>\app\build.gradle or the Module Build File is for a specific module so it will be used for specific module level configs. A Module Build File contains the following:

android settings

compileSdkVersion

buildToolsVersion

defaultConfig and productFlavors

manifest properties such as applicationId, minSdkVersion, targetSdkVersion, and test information

buildTypes

build properties such as debuggable, ProGuard enabling, debug signing, version name suffix and testinformation

dependencies

==============================================================================================================

==============================================================================================================

Android Plug-in for Gradle

In this document

Work with build variants

See also

Build System Overview

Building and Running

Building and Running from Android Studio

Download

PLUGIN LANGUAGE REFERENCE

android-gradle-plugin-dsl.zip

The Android build system consists of an Android plugin for Gradle. Gradle is an advanced build toolkit that manages dependencies and allows you to define custom build logic. Android Studio uses a Gradle wrapper to fully integrate the Android plugin for Gradle. The Android plugin for Gradle also runs independent of Android Studio. This means that you can build your Android apps from which Android Studio and from the command line on your machine or on machines where Android Studio is not installed (such as continuous integration servers).

The output of the build is the same whether you are building a project from the command line, on a remote machine, or using Android Studio.

Build configuration

The build configuration for your project is defined inside build.gradle files, which are plain text files that use the syntax and options from Gradle and the Android plugin to configure the following aspects of your build:

Build variants. The build system can generate multiple APKs with different product and build configurations for the same module. This is useful when you want to build different versions of your application without having to create a separate projects or modules for each version.

Dependencies. The build system manages project dependencies and supports dependencies from your local filesystem and from remote repositories. This prevents you from having to search, download, and copy binary packages for your dependencies into your project directory.

Manifest entries. The build system enables you to specify values for some elements of the manifest file in the build variant configuration. These build values override the existing values in the manifest file. This is useful if you want to generate multiple APKs for your modules where each of the apk files has a different application name, minimum SDK version, or target SDK version. When multiple manifests are present, manifest settings are merged in priority of buildType and productFlavor, /main manifest, and the library manifests.

Signing. The build system enables you to specify signing settings in the build configuration, and it can sign your APKs during the build process.

ProGuard. The build system enables you to specify a different ProGuard rules file for each build variant. The build system can run ProGuard to obfuscate your classes during the build process.

Testing. For most templates, the build system creates a test directory, androidTest and generates a test APK from the test sources in your project, so you do not have to create a separate test project. The build system can also run your tests during the build process.

Gradle build files use Domain Specific Language (DSL) to describe and manipulate the build logic through Groovy syntax. Groovy is a dynamic language that you can use to define custom build logic and to interact with the Android-specific elements provided by the Android plugin for Gradle.

Build by convention

The Android Studio build system assumes sensible defaults for the project structure and other build options. If your project adheres to these conventions, your Gradle build files are very simple. When some of these conventions do not apply to your project, the flexibility of the build system allows you to configure almost every aspect of the build process. For example, if you need to replace the default source folders in your module directories, you can configure a new directory structure in the module's build file.

Projects and modules build settings

A project in Android Studio represents the top-level Android development structure. Android Studio projects contain project files and one or more application modules. A module is a component of your app that you can build, test, or debug independently. Modules contain the source code and resources for your apps. Android Studio projects can contain several kinds of modules:

Android application modules contain application (mobile, TV, Wear, Glass) code and may depend on library modules, although many Android apps consists of only one application module. The build system generates APK packages for application modules.

Android library modules contain reusable Android-specific code and resources. The build system generates an AAR (Android ARchive) package for library modules.

App Engine modules contain code and resources for App Engine integration.

Java library modules contain reusable code. The build system generates a JAR package for Java library modules.

Android Studio projects contain a top-level project Gradle build file that allows you to add the configuration options common to all application modules in the project. Each application module also has its own build.gradle file for build settings specific to that module.

Project Build File

By default, the project-level Gradle file uses buildscript to define the Gradle repositories and dependencies. This allows different projects to use different Gradle versions. Supported repositories include JCenter, Maven Central, or Ivy. This example declares that the build script uses the JCenter repository and a classpath dependency artifact that contains the Android plugin for Gradle version 1.0.1.

buildscript {

repositories {

jcenter()

}

dependencies {

classpath 'com.android.tools.build:gradle:1.0.1'

// NOTE: Do not place your application dependencies here: they belong

// in the individual module build.gradle files

}

}

allprojects {

repositories {

jcenter()

}

}

Note: The SDK location for the Android Studio project is defined in the local.properties file in the sdk.dir setting or through an ANDROID\_HOME environment variable.

Module Build File

The application module Gradle build file allows you to configure module build settings, including overriding the src/main manifest settings and setting custom packaging options.

android settings

compileSdkVersion

buildToolsVersion

defaultConfig and productFlavors

manifest properties such as applicationId, minSdkVersion, targetSdkVersion, and test information

buildTypes

build properties such as debuggable, ProGuard enabling, debug signing, version name suffix and testinformation

dependencies

This example applies the Android plugin, uses the default configuration to override several manifest properties, creates two build types: release and debug, and declares several dependencies.

apply plugin: 'com.android.application'

android {

compileSdkVersion 20

buildToolsVersion "20.0.0"

defaultConfig {

applicationId "com.mycompany.myapplication"

minSdkVersion 13

targetSdkVersion 20

versionCode 1

versionName "1.0"

}

buildTypes {

release {

minifyEnabled false

proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'

}

debug {

debuggable true

}

}

}

dependencies {

compile fileTree(dir: 'libs', include: ['\*.jar'])

compile 'com.android.support:appcompat-v7:20.0.0'

compile project(path: ':app2, configuration: 'android-endpoints')

}

Note: You can inject custom build logic for property values defined by a function that gets called by the property, for example:

def computeVersionName() {

...

}

android {

defaultConfig {

versionName computeVersionName()

...

}

}

Dependencies

The Android Studio build system manages project dependencies and supports module dependencies, local binary dependencies, and remote binary dependencies.

Module Dependencies

An application module can include in its build file a list of other modules it depends on. When you build this module, the build system assembles and includes the required modules.

Local Dependencies

If you have binary archives in your local filesystem that a module depends on, such as JAR files, you can declare these dependencies in the build file for that module.

Remote Dependencies

When some of your dependencies are available in a remote repository, you do not have to download them and copy them into your project. The Android Studio build system supports remote dependencies from repositories, such as Maven, and dependency managers, such as Ivy.

Many popular software libraries and tools are available in public Maven repositories. For these dependencies you only have to specify their Maven coordinates, which uniquely identify each element in a remote repository. The format for Maven coordinates used in the build system is group:name:version. For example, the Maven coordinates for version 16.0.1 of the Google Guava libraries are com.google.guava:guava:16.0.1.

The Maven Central Repository is widely used to distribute many libraries and tools.

Build tasks

The Android Studio build system defines a hierarchical set of build tasks: the top-level or anchor tasks invoke dependent tasks to produce their collective build outcomes. The top-level build tasks are:

assemble

Builds the project output.

check

Runs checks and tests.

build

Runs both assemble and check.

clean

Performs the clean.

The Android plugin provides the connectedCheck and deviceCheck tasks for checks run on connected, emulated, and remote devices. Gradle tasks can be viewed by clicking the Gradle tab in the right margin.

You can view the list of available tasks and invoke any task from Android Studio and from the command line, as described in Building and Running from Android Studio and Build the project from the command line.

The Gradle wrapper

Android Studio projects contain the Gradle wrapper, which consists of:

A JAR file

A properties file

A shell script for Windows platforms

A shell script for Mac and Linux platforms

Note: You should submit all of these files to your source control system.

Using the Gradle wrapper (instead of the local Gradle installation) ensures that you always run the version of Gradle defined in the local.properties file. To configure your project to use a newer version of Gradle, edit the properties file and specify the new version there.

Android Studio reads the properties file from the Gradle wrapper directory inside your project and runs the wrapper from this directory, so you can seamlessly work with multiple projects that require different versions of Gradle.

Note: Android Studio does not use the shell scripts, so any changes you make to them won't work when building from the IDE. You should define your custom logic inside Gradle build files instead.

You can run the shell scripts to build your project from the command line on your development machine and on other machines where Android Studio is not installed.

Caution: When you create a project, only use the Gradle wrapper scripts and JAR from a trusted source, such as those generated by Android Studio.

Build variants

Each version of your app is represented in the build system by a build variant. Build variants are combinations of product flavors and build types. Product flavors represent product build versions of an app, such as free and paid. Build types represent the build packaging versions generated for each app package, such as debug and release. The build system generates APKs for each combination of product flavor and build type.

By default, Android Studio defines default configuration settings, defaultConfig in the build.gradle file, and two build types (debug and release). This creates two build variants, debug and release, and the build system generates an APK for each variant.

Adding two product flavors, demo and full along with the default build types debug and release generates four build variants, each with its own customized configuration:

demoDebug

demoRelease

fullDebug

fullRelease

Resources are merged across the multiple Android application sources:

Build variants based on the buildType, and productFlavor build settings

The main sourceSet, generally located in src/main/res

Library Project dependencies, which contribute resources through the res entry in their aar bundle.

The priority of the merge order from lowest to highest is libraries/dependencies -> main src -> productFlavor -> buildType.

Some projects have complex combinations of features along more than one dimension, but they still represent the same app. For example, in addition to having a demo and a full version of the app, some games may contain binaries specific to a particular CPU/ABI. The flexibility of the build system makes it possible to generate the following build variants for such a project:

x86-demoDebug

x86-demoRelease

x86-fullDebug

x86-fullRelease

arm-demoDebug

arm-demoRelease

arm-fullDebug

arm-fullRelease

mips-demoDebug

mips-demoRelease

mips-fullDebug

mips-fullRelease

This project would consist of two build types (debug and release) and two dimensions of product flavors, one for app type (demo or full) and one for CPU/ABI (x86, ARM, or MIPS).

Source directories

To build each version of your app, the build system combines source code and resources from:

src/main/ - the main source directory (the default configuration common to all variants)

src/<buildType>/ - the source directory

src/<productFlavor>/ - the source directory

Note: The build type and product flavor source directories are optional, as Android Studio does not create these directories for you. You should create these directories as you add build types and product flavors to the build configuration files. The build system does not use these directories if they are not present.

For projects that do not define any flavors, the build system uses the defaultConfig settings, the main app directory and the default build type directories. For example, to generate the default debug and release build variants in projects with no product flavors, the build system uses:

src/main/ (default configuration)

src/release/ (build type)

src/debug/ (build type)

For projects that define a set of product flavors, the build system merges the build type, product flavor and main source directories. For example, to generate the full-debug build variant, the build system merges the build type, product flavor and main directories:

src/main/ (default configuration)

src/debug/ (build type)

src/full/ (flavor)

For projects that use flavor dimensions, the build system merges one flavor source directory per dimension. For example, to generate the arm-demo-release build variant, the build system merges:

src/main/ (default configuration)

src/release/ (build type)

src/demo/ (flavor - app type dimension)

src/arm/ (flavor - ABI dimension)

The source code from these directories is used together to generate the output for a build variant. You can have classes with the same name in different directories as long as those directories are not used together in the same variant.

The build system also merges all the manifests into a single manifest, so each build variant can define different components or permissions in the final manifest. The manifest merge priority from lowest to highest is libraries/dependencies -> main src -> productFlavor -> buildType.

The build system merges all the resources from the all the source directories. If different folders contain resources with the same name for a build variant, the priority order is the following: build type resources override those from the product flavor, which override the resources in the main source directory, which override those in any libraries.

Note: Build variants enable you to reuse common activities, application logic, and resources across different versions of your app.

Execution faild for :dexDebug

=============================

The right answer is, that some of your jar files does not compile. You should go into your build.gradle file in your project, and look in your dependencies.

If you're just importing some jar files, you could try to remove them and add them one at a time. This will help you determine which one of them causes the error.

In my case, I did just that, and when I was importing the last one, the app compiled. So I think the real problem was that I was importing too many at once. But now it all works.