Building Apps with Over 65K Methods

As the Android platform has continued to grow, so has the size of Android apps. When your application and the libraries it references reach a certain size, you encounter build errors that indicate your app has reached a limit of the Android app build architecture. Earlier versions of the build system report this error as follows:

```
Conversion to Dalvik format failed:
Unable to execute dex: method ID not in [0, 0xffff]: 6553
```

More recent versions of the Android build system display a different error, which is an indication of the same problem:

```
trouble writing output:
Too many field references: 131000; max is 65536.
You may try using --multi-dex option.
```

Both these error conditions display a common number: 65,536. This number is significant in that it represents the total number of references that can be invoked by the code within a single Dalvik Executable (dex) bytecode file. If you have built an Android app and received this error, then congratulations, you have a lot of code! This document explains how to move past this limitation and continue building your app.

> **Note:** The guidance provided in this document supersedes the guidance given in the Android Developers blog post Custom Class Loading in Dalvik (http://android-developers.blogspot.com/2011/07/custom-class-loading-in-dalvik.html).

About the 65K Reference Limit

Android application (APK) files contain executable bytecode files in the form of Dalvik (https://source.android.com/devices/tech/dalvik/) Executable (DEX) files, which contain the compiled code used to run your app. The Dalvik Executable specification limits the total number of methods that can be referenced within a single DEX file to 65,536, including Android framework methods, library methods, and methods in your own code. Getting past this limit requires that you configure your app build process to generate more than one DEX file, known as a *multidex* configuration.

Multidex support prior to Android 5.0

Versions of the platform prior to Android 5.0 use the Dalvik runtime for executing app code. By default, Dalvik limits apps to a single classes.dex bytecode file per APK. In order to get around this limitation, you can use the multidex support library (/tools/support-library/features.html#multidex), which becomes part of the primary DEX file of your app and then manages access to the additional DEX files and the code they contain.

Multidex support for Android 5.0 and higher

Android 5.0 and higher uses a runtime called ART which natively supports loading multiple dex files from application APK files. ART performs pre-compilation at application install time which scans for classes(..N).dex files and compiles them into a single .oat file for execution by the Android device. For more information on the Android 5.0 runtime, see Introducing ART (https://source.android.com/devices/tech/dalvik/art.html).

Avoiding the 65K Limit

Before configuring your app to enable use of 65K or more method references, you should take steps to reduce the total number of references called by your app code, including methods defined by your app code or included libraries. The following strategies can help you avoid hitting the dex reference limit:

- **Review your app's direct and transitive dependencies** - Ensure any large library dependency you include in your app is used in a manner that outweighs the amount of code being added to the application. A common anti-pattern is to include a very large library because a few utility methods were useful. Reducing your app code dependencies can often help you avoid the dex reference limit.

- **Remove unused code with ProGuard** - Configure the ProGuard settings for your app to run ProGuard and ensure you have shrinking enabled for release builds. Enabling shrinking ensures you are not shipping unused code with your APKs.

Using these techniques can help you avoid the build configuration changes required to enable more method references in your app. These steps can also decrease the size of your APKs, which is particularly important for markets where bandwidth costs are high.

Configuring Your App for Multidex with Gradle

The Android plugin for Gradle available in Android SDK Build Tools 21.1 and higher supports multidex as part of your build configuration. Make sure you update the Android SDK Build Tools tools and the Android Support Repository to the latest version using the SDK Manager (/tools/help/sdk-manager.html) before attempting to configure your app for multidex.

Setting up your app development project to use a multidex configuration requires that you make a few modifications to your app development project. In particular you need to perform the following steps:

- Change your Gradle build configuration to enable multidex
- Modify your manifest to reference the `MultiDexApplication` class

Modify your app Gradle build file configuration to include the support library and enable multidex output, as shown in the following Gradle build file snippet:

```
android {
    compileSdkVersion 21
    buildToolsVersion "21.1.0"

    defaultConfig {
        ...
        minSdkVersion 14
        targetSdkVersion 21
        ...

        // Enabling multidex support.
        multiDexEnabled true
    }
    ...
}

dependencies {
  compile 'com.android.support:multidex:1.0.0'
}
```

> **Note:** You can specify the `multiDexEnabled` setting in the `defaultConfig`, `buildType`, or `productFlavor` sections of your Gradle build file.

In your manifest add the `MultiDexApplication` `(/reference/android/support/multidex/MultiDexApplication.html)` class from the multidex support library to the application element.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.android.multidex.myapplication">
    <application
        ...
        android:name="android.support.multidex.MultiDexApplication">
        ...
    </application>
</manifest>
```

When these configuration settings are added to an app, the Android build tools construct a primary dex (classes.dex) and supporting (classes2.dex, classes3.dex) as needed. The build system will then package them into an APK file for distribution.

> **Note:** If your app uses extends the `Application` (/reference/android/app/Application.html) class, you can override the attachBaseContext() method and call MultiDex.install(this) to enable multidex. For more information, see the `MultiDexApplication` (/reference/android/support/multidex/MultiDexApplication.html) reference documentation.

Limitations of the multidex support library

The multidex support library has some known limitations that you should be aware of and test for when you incorporate it into your app build configuration:

- The installation of .dex files during startup onto a device's data partition is complex and can result in Application Not Responding (ANR) errors if the secondary dex files are large. In this case, you should apply code shrinking techniques with ProGuard to minimize the size of dex files and remove unused portions of code.
- Applications that use multidex may not start on devices that run versions of the platform earlier than Android 4.0 (API level 14) due to a Dalvik linearAlloc bug (Issue 22586). If you are targeting API levels earlier than 14, make sure to perform testing with these versions of the platform as your application can have issues at startup or when particular groups of classes are loaded. Code shrinking can reduce or possibly eliminate these potential issues.
- Applications using a multidex configuration that make very large memory allocation requests may crash during run time due to a Dalvik linearAlloc limit (Issue 78035). The allocation limit was increased in Android 4.0 (API level 14), but apps may still run into this limit on Android versions prior to Android 5.0 (API level 21).
- There are complex requirements regarding what classes are needed in the primary dex file when executing in the Dalvik runtime. The Android build tooling updates handle the Android requirements, but it is possible that other included libraries have additional dependency requirements including the use of introspection or invocation of Java methods from native code. Some libraries may not be able to be used until the multidex build tools are updated to allow you to specify classes that must be included in the primary dex file.

Optimizing Multidex Development Builds

A multidex configuration requires significantly increased build processing time because the build system must make complex decisions about what classes must be included in the primary DEX file and what classes can be included in secondary DEX files. This means that routine builds performed as part of the development process with multidex typically take longer and can potentially slow your development process.

In order to mitigate the typically longer build times for multidex output, you should create two variations on your build output using the Android plugin for Gradle `productFlavors` (http://tools.android.com/tech-docs/new-build-system/user-guide#TOC-Product-flavors): a development flavor and a production flavor.

For the development flavor, set a minimum SDK version of 21. This setting generates multidex output much faster using the ART-supported format. For the release flavor, set a minimum SDK version which matches your actual minimum support level. This setting generates a multidex APK that is compatible with more devices, but takes longer to build.

The following build configuration sample demonstrates the how to set up these flavors in a Gradle build file:

```
android {
    productFlavors {
        // Define separate dev and prod product flavors.
        dev {
            // dev utilizes minSDKVersion = 21 to allow the Android gradle plugin
            // to pre-dex each module and produce an APK that can be tested on
            // Android Lollipop without time consuming dex merging processes.
            minSdkVersion 21
        }
        prod {
            // The actual minSdkVersion for the application.
            minSdkVersion 14
        }
    }
          ...
    buildTypes {
        release {
            runProguard true
            proguardFiles getDefaultProguardFile('proguard-android.txt'),
                                                 'proguard-rules.pro'
        }
    }
}
dependencies {
  compile 'com.android.support:multidex:1.0.0'
}
```

After you have completed this configuration change, you can use the `devDebug` variant of your app, which combines the attributes of the `dev` productFlavor and the `debug` buildType. Using this target creates a debug app with proguard disabled, multidex enabled, and minSdkVersion set to Android API level 21. These settings cause the Android gradle plugin to do the following:

1. Build each module of the application (including dependencies) as separate dex files. This is commonly referred to as pre-dexing.
2. Include each dex file in the APK without modification.
3. Most importantly, the module dex files will not be combined, and so the long-running calculation to determine the contents of the primary dex file is avoided.

These settings result in fast, incremental builds, because only the dex files of modified modules are recomputed and repackaged into the APK file. The APK that results from these builds can be used to test on Android 5.0 devices only. However, by implementing the configuration as a flavor, you preserve the ability to perform normal builds with the release-appropriate minimum SDK level and proguard settings.

You can also build the other variants, including a `prodDebug` variant build, which takes longer to build, but can be used for testing outside of development. Within the configuration shown, the `prodRelease` variant would be the final testing and release version. If you are executing gradle tasks from the command line, you can use standard commands with `DevDebug` appended to the end (such as `./gradlew installDevDebug`). For more information about using flavors with Gradle tasks, see the Gradle Plugin User Guide (http://tools.android.com/tech-docs/new-build-system/user-guide).

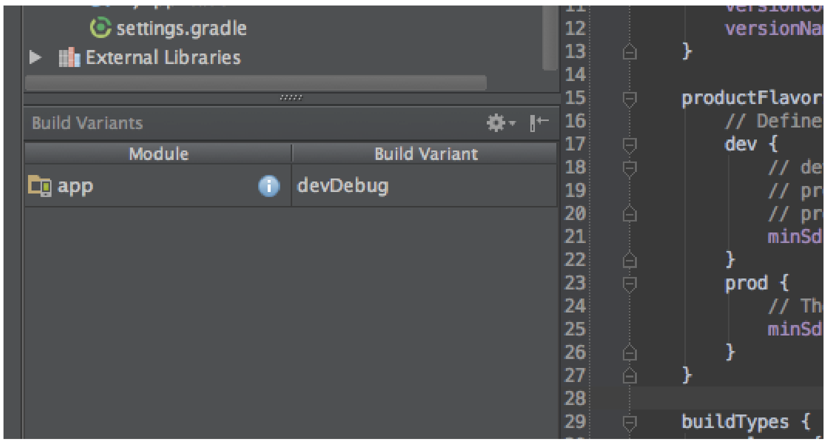
Tip: You can also provide a custom manifest, or a custom application class for each flavor, allowing you to use the support library MultiDexApplication class, or calling MultiDex.install() only for the variants that need it.

Using Build Variants in Android Studio

Build variants can be very useful for managing the build process when using multidex. Android Studio allows you to select these build variants in the user interface.

To have Android Studio build the "devDebug" variant of your app:

1. Open the *Build Variants* window from the left-sidebar. The option is located next to *Favorites*.
2. Click the name of the build variant to select a different variant, as shown in Figure 1.



Module	Build Variant
app	devDebug

```
        versionNa
    }

    productFlavor
        // Define
        dev {
            // de
            // pr
            // pr
            minSd
        }
        prod {
            // Th
            minSd
        }
    }

    buildTypes {
```

Figure 1. Screen shot of the Android Studio left panel showing a build variant.

Note: The option to open this window is only available after you have successfully synchronized Android Studio with your Gradle build file using the **Tools > Android > Sync Project with Gradle Files** command.

Testing Multidex Apps

Testing apps that use multidex configuration require some additional steps and configuration. Since the location of code for classes is not within a single DEX file, instrumentation tests do not run properly unless configured for multidex.

When testing a multidex app with instrumentation tests, use MultiDexTestRunner

(/reference/com/android/test/runner/MultiDexTestRunner.html) from the multidex testing support library. The following sample `build.gradle` file, demonstrates how to configure your build to use this test runner:

```
android {
  defaultConfig {
      ...
      testInstrumentationRunner "android.support.multidex.MultiDexTestRunner"
  }
}

dependencies {
    androidTestCompile 'com.android.support:multidex-instrumentation:1.0.0'
}
```

You may use the instrumentation test runner class directly or extend it to fit your testing needs. Alternatively, you can override onCreate in existing instrumentations like this:

```
public void onCreate(Bundle arguments) {
    MultiDex.install(getTargetContext());
    super.onCreate(arguments);
    ...
}
```

Note: Use of multidex for creating a test APK is not currently supported.