AAPT

(Android Asset Packaging tool)

This tool is part of the SDK (and build system) and allows you to view, create, and update Zip-compatible archives (zip, jar, apk). It can also compile resources into binary assets.

Build scripts and IDE plugins utilise this tool to package the apk file that constitutes an Android application.

Path

sdk/build-tools/{any version}/

list
The 'list' command shows the contents of the package.

aapt list app-debug.apk

the -v option shows the contents of the zipped archive, and is just like 'unzip -l -v'

aapt list -v app-debug.apk

Understanding Android Runtime and Dalvik

JVM

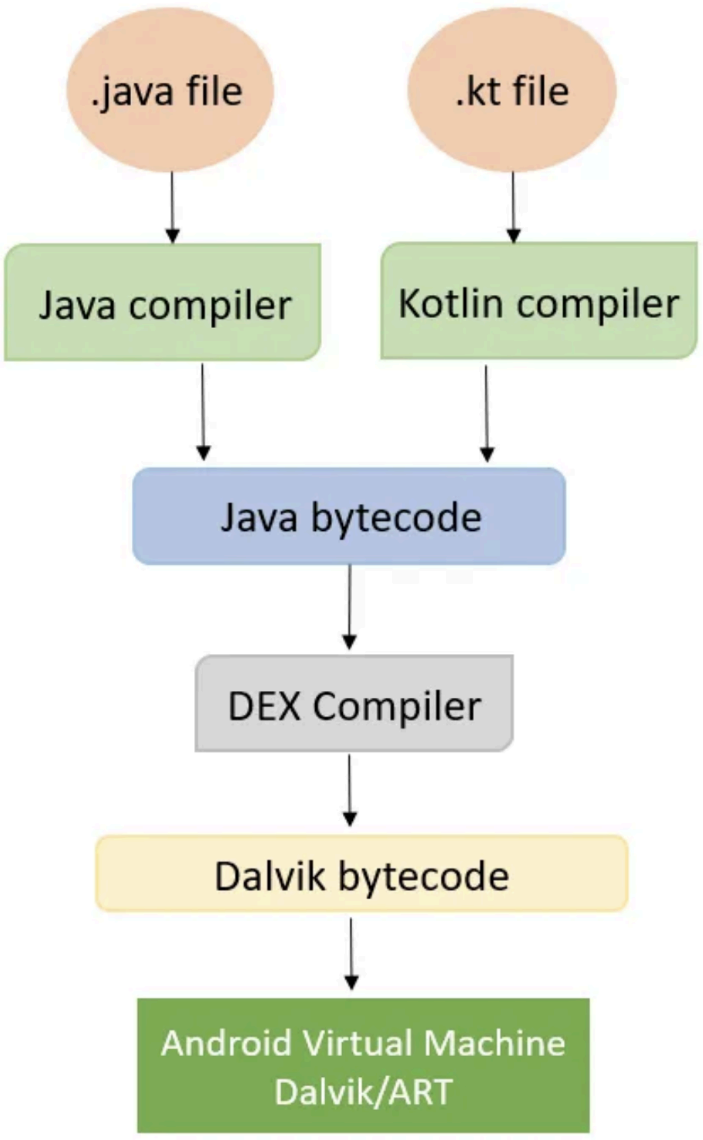
Java Virtual Machine is an engine that enables a computer to run applications that have been compiled to Java bytecode. So at its core, JVM converts java bytecode to machine code.

So why is JVM not used for Android?
JVM is designed for powerful systems with large amounts of storage. But in the initial days of Android, storage and performance were not its strongest suites. The first android phone barely touched 200 MB of RAM. Clearly JVM was not an option here. To solve this problem, Google came up with Dalvik Virtual Machine.

DVM

The Dalvik Virtual Machine is optimized for memory, power and battery usage to work efficiently on android devices.

But DVM cannot work directly with Java bytecode generated by a Java or a Kotlin compiler. The Java bytecode needs to be converted into Dalvik bytecode. DVM then converts this Dalvik bytecode to machine code.



Primary concern of Dalvik was RAM optimization. So instead of converting the whole application bytecode to machine code in the beginning, it used the strategy called JIT or Just In Time compilation. In this strategy, the compiler compiles only parts of code that are needed at runtime during execution of the app. Since the entire code is not compiled at once, RAM is optimized but it results in a major impact on the application's runtime performance. To tackle this problem, Google introduced ART — Android Runtime.

ART

As the hardware capabilities of Android phones improved, developers started to question the dynamic compilation strategy. Introduced in API 21 (Android L), ART used an opposite compilation strategy. Instead of using JIT, it used AOT or Ahead of Time compilation. As the name suggests, now the dex bytecode was compiled to machine code, which was saved in a .oat file before running the app. Every time the app was opened, ART loaded this .oat file eliminating the need for JIT compilation. It improved runtime performance significantly, but had 2 major drawbacks:

- ☐ Higher RAM consumption
- ☐ Longer app install and update times. (as app’s dex byte code is converted to machine code during installation)

Soon Google realized that most parts of the application are rarely used by the majority of users. Clearly, it was inefficient to hold the entire app’s machine code in memory, if most of it remained unused.

Profile guided compilation & re-emergence of JIT

In API 23 (Nougat), Google re-introduced JIT but with a tweak called Profile-guided compilation. This is a strategy that allows improving app performance with time as they run over and over. Default compilation strategy is JIT, but when ART detects that some methods are used more frequently than others, or are “hot”, ART will use AOT strategy to precompile and then cache the frequently used hot methods so that these don’t have to be compiled every time (This pre-compilation was done only when the device was charging and idle). This approach had a few advantages:

RAM optimization — Since the majority of app’s code is not used frequently, only a small amount of code is precompiled and held in RAM.

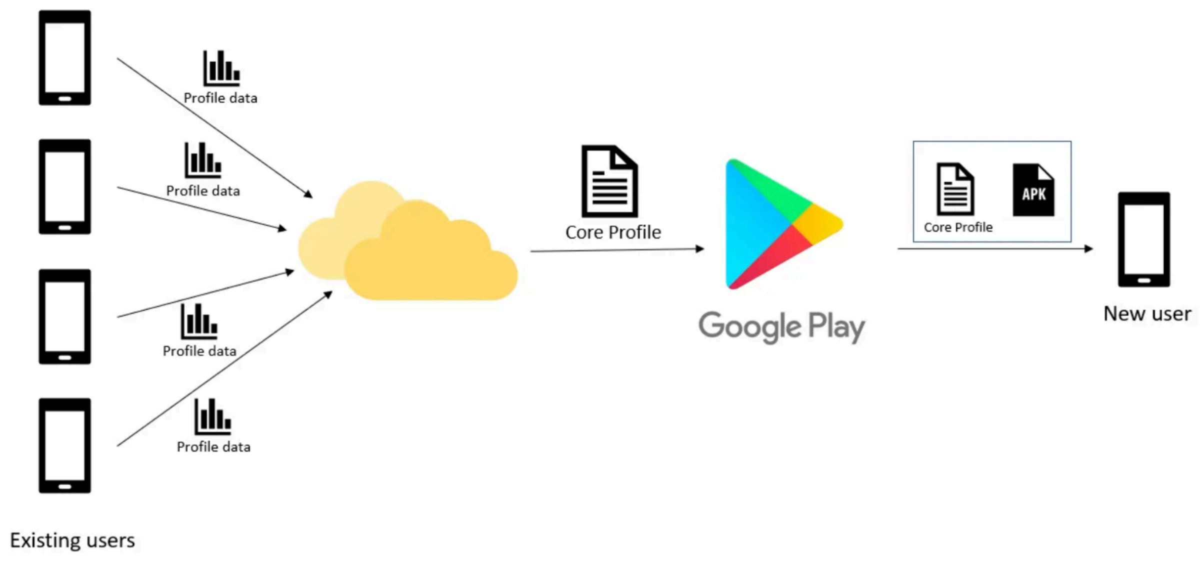
Faster App installs — During installation, there is no data for app usage. So during installation, ART does not know which parts of bytecode to convert to machine code.

This worked great, but still had a problem. To accumulate data about “hot” methods, the app needed to run a few times. Obviously, these initial few runs of the app would mostly be JIT based, and therefore, slow. To fix this Google introduced Profiles in the cloud.

Profiles in the cloud

The above discussed solution would have been perfect if somehow the profile data about the most frequently used methods could be downloaded along with the app’s apk. This is exactly what Google did with Profiles in the cloud. Introduced in API 28 (Android Pie), this strategy leveraged the fact that most users use a particular application in a more or less similar fashion. Data about frequently used code parts is collected from the existing users of this application. This collected data is used to create a common core profile for this application.

Now when a new user downloads the app from Google play, this core profile is also downloaded. ART uses this file to pre-compile the frequently used code parts. As the user uses the app with time, ART collects profile data for this user and re-compiles code parts that are frequently used by this particular user.



Profiles in the cloud

Deep dive into Android build process

Android build process entails several tools and steps that convert a project into an APK (Android Application Package) or an AAB (Android App Bundle). Let’s dive into these steps.

Step 1: Resource compilation

Every Android developer has written the following line of code at

some point or the other:

```
//Java
Button b = findViewById(R.id.sample_button);
//Kotlin
var b : Button = findViewById(R.id.sample_button);
```

Ever wondered where this “R” class comes from? R.java is the love child of Android Asset Packaging Tool (AAPT2) and your project’s res/ directory. This tool takes the AndroidManifest.xml and all the resources under res/directory and parses, indexes, and compiles them to create the R class which has all the resource ids

Step 2: Source code compilation

In this step, all your Kotlin and Java files (including the R.java file generated in step 1 and the code from the app’s dependencies and libraries) are compiled into .class files by kotlinc and javac compilers respectively. Your project’s aidl files are converted to Java interfaces and then subsequently compiled to .class files.

But this bytecode is not any use in the .class form. Android apps do not run on JVM. They run on Android Runtime (ART) and pre Lollipop on Dalvik Virtual Machine (DVM). These environments do not understand .class files. They only understand DEX files (Dalvik EXecutable)

Step 3: Shrinking, obfuscation and optimization

This step is optional. Build process will go through this step only if you have enabled shrinking in your release build like this:

```
android {
   buildTypes {
      release {
         minifyEnabled true
         shrinkResources true
         proguardFiles getDefaultProguardFile(
               'proguard-android-optimize.txt'),
               'proguard-rules.pro'
      }
   }
   ...
}
```

This step involves the following sub steps:

Code shrinking — also known as tree-shaking, detects and safely removes unused classes, fields, methods, and attributes from your app and its library dependencies.

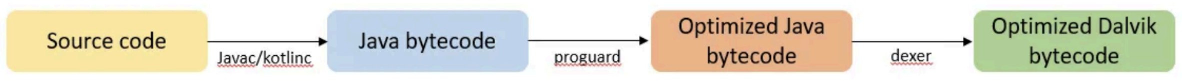
Resource shrinking — removes unused resources including the ones in the app’s library dependencies.

Obfuscation — replaces the original names of classes and members with short meaningless names. This results in reduced DEX file sizes.

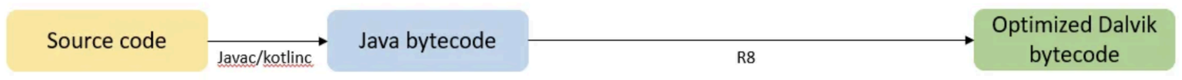
Optimization — inspects and removes unnecessary code. For instance, if your else{} block cannot be reached in any case, it will be removed in this step.

Two main tools to execute this step are R8 and ProGuard. When you build your project using Android Gradle plugin 3.4.0 or higher, the plugin no longer uses ProGuard to perform compile-time code optimization. Instead, it uses the R8 compiler. You can choose to modify the default behaviour of R8 using the ProGuard rules files. R8 offers the following advantages when compared to ProGuard:

- Proguard reduces the app size by 8.5% whereas R8 reduces it by 10%.
- R8 offers better, more extensive Kotlin support.
- In the case of ProGuard, it first converted java bytecode to optimised java bytecode, which was subsequently converted to DEX files by a dexer. But R8 directly converts java bytecode to optimized DEX code.



ProGuard



R8

Step 4: Dalvik bytecode (DEX files) generation

This step is executed within step 3 IF step 3 is executed.
At this juncture, our .class files are converted into .dex files or Dalvik EXecutable files. This job is done by D8 tool or the R8 tool if you have enabled code optimization as discussed in step 3.

D8 is a dexer that converts .class files to .dex files that can be understood by ART. It also enables you to use Java 8 features in your code through a compile process called **desugaring**.

In general, desugaring refers to the process of providing backward compatibility for new Java libraries.

ART does not support Java 8 language features by default. Desugaring essentially converts Java 8 bytecode to Java 7 bytecode. D8 not only converts our application’s .class files to .dex files, but also adds dex code for the newer java libraries and then bundles them together in the APK.

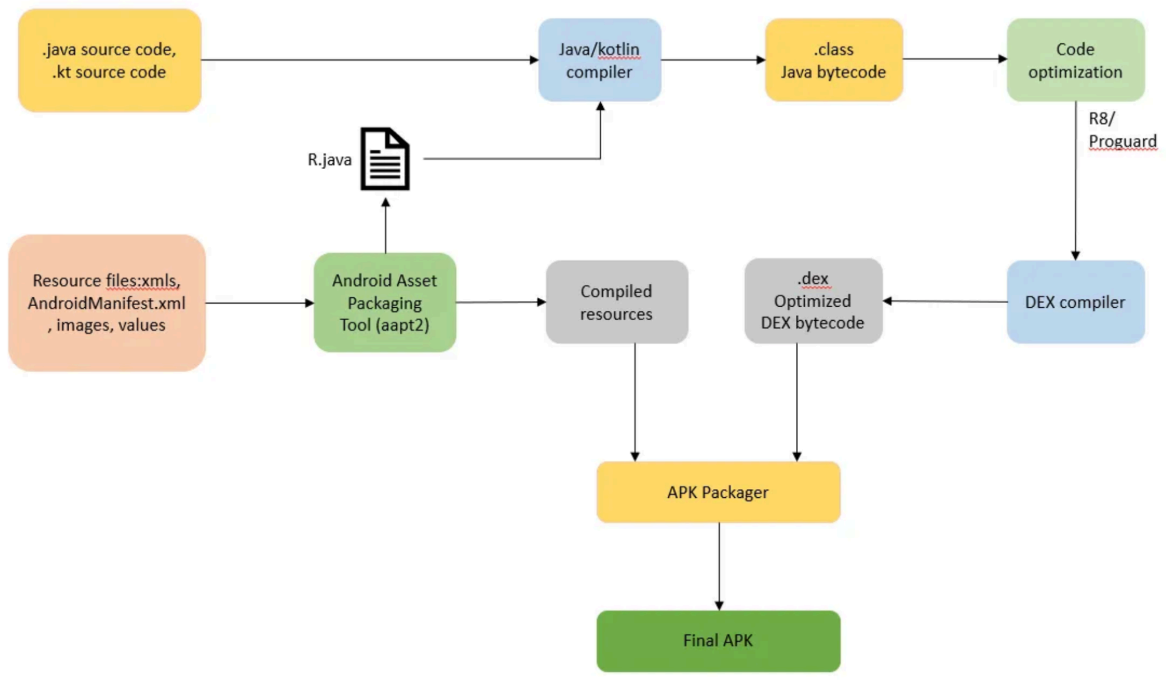
Step 5: Packaging

The packager combines the DEX files from step 3/step4 and the compiled resources from step 1 into an APK or an AAB. Before generating your final APK, the packager uses the **zipalign** tool to optimize your app to use less memory when running on a device. zipalign is a zip archive alignment tool. It ensures that all uncompressed files in the archive are aligned relative to the start of the file. This allows those files to be accessed directly, removing the need to copy this data in RAM and reducing your app's memory usage.

Before your app can be installed, this APK/AAB must be signed.

- If it is a debug version of your app, packager signs your app with the debug keystore, and you don’t need to do anything.

- If it is a release version of your app, the packager signs your app with the release keystore that you need to configure. For more details about creating a release keystore, read about signing your app in Android Studio.



Android build process