Android Runtime

This guide includes all the steps involved in creating an Android app: from the source code all the way up to a user-ready app.

First Step : Compile

In the beginning we have the source code for the app, which is either java or kotlin. The first step is to compile the source code to ByteCode by Compiler.
This ByteCode is stored with a specific format. For these files, the extension is (.class) and that is why these files are called “class files”.

Second Step : Dexing or DEX Conversion

In this step, the ByteCode is converted to Dalvik code by DEX Compiler.

What is the difference between ByteCode and Dalvik code ?

First let’s take a look at why we need Dalvik code. Java works by the principle “write once, run everywhere”. This means we can type the code just one time and then run it on any operating system, like Windows or Linux, seamlessly. This is possible because these operating systems include JVM (Java Virtual Machine), which converts ByteCode to machine code. After that, the cpu can execute the machine code.
In the Android operating system, however, there is no JVM. Instead, there is a similar tool called DVM (Dalvik Virtual Machine). Since DVM does not work with ByteCode, we must convert it to Dalvik code.

Suppose we have a short java code like this :

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

In the first step it will convert to ByteCode:

```
Compiled from "HelloWorld.java"
public class HelloWorld {
  public HelloWorld();
    Code:
       0: aload_0
       1: invokespecial #1
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #2
       3: ldc           #3
       5: invokevirtual #4
       8: return
}
```

In the second step it will convert to Dalvik code:

```
.class public LHelloWorld;
.super Ljava/lang/Object;

.method public constructor <init>()V
    .registers 1
    invoke-direct {p0}, Ljava/lang/Object;-><init>()V
    return-void
.end method

.method public static main([Ljava/lang/String;)V
    .registers 2
    sget-object v0, Ljava/lang/System;->out:Ljava/io/PrintStream;
    const-string v1, "Hello World!"
    invoke-virtual {v0, v1}, Ljava/io/PrintStream;->println(Ljava/lang/String;)V
    return-void
.end method
```

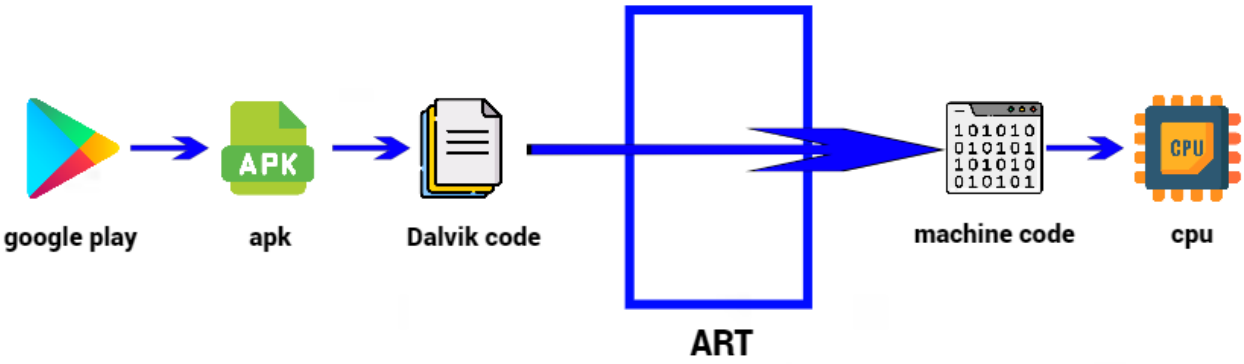
Third Step : Packaging

In the third step, we use AAPT (Android Asset Packaging Tool). Here the Dalvik code and all associated files (images, XML files, etc,) are packaged into APK/AAB.
XML files don't compile. They will first be converted to binary XML format in order to be packaged within apk.

Final Step : publish

We can now publish apk on Google Play where it will be accessible to all users.

Next we will talk about the steps involved when a user installs the app on Google Play



First Step : Decompress

In this step apk reverts back to Dalvik code (a Dex file).

Second Step : Android Runtime (ART)

What is the ART?

ART (Android Runtime) is the step responsible for compiling DEX files to machine code.
Over the years, different Android versions employed distinct strategies for this task.
So let's explore each strategy:

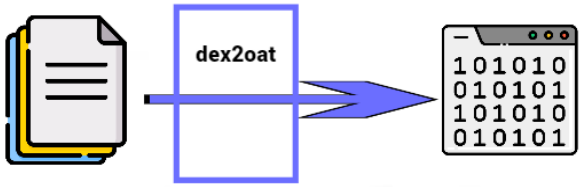
First Strategy : Just In Time (JIT)

This strategy was used from Android 1 through Android 4 (Kit Kat).
In the past, memory was very limited so it was difficult to compile the whole app to machine code. As a result, we used the principle Just In time Compilation (JIT). JIT means we compile only a specific part of the code while we run the app.
Dalvik compiled part of the code so that it would save memory, yet there were a lot of performance problems. Here's why. Sometimes a certain portion of code is repeated.

Instead of compiling this portion of code once, Dalvik was compiling it each time it appeared within the code. In order to make this more efficient, Android made caches for these repeated codes. Due to limited memory, however, the problems persisted.

Second Strategy : Ahead Of Time (AOT)

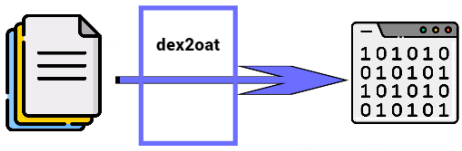
This strategy was used in Android 5 (lollipop) and Android 6 (marshmallow)
In this strategy we used the principle Ahead Of Time (AOT). AOT would compile the whole app during installation. Here, the DEX files compile to native machine code using a special tool called DEX2OAT.



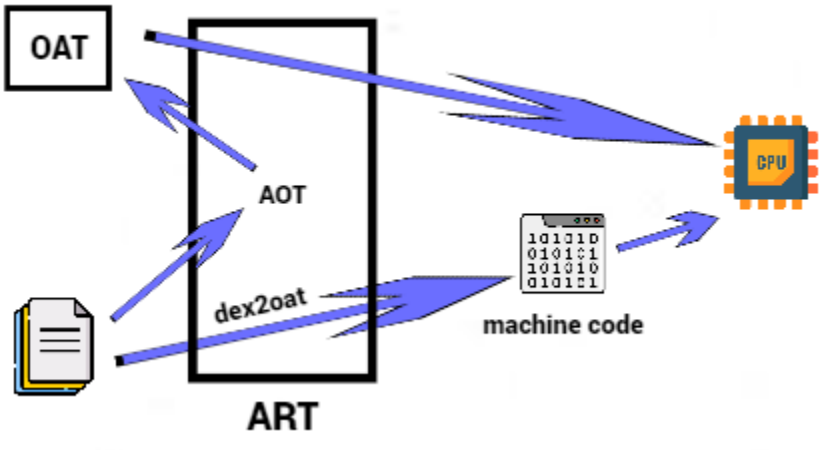
This strategy improved the performance but still had problems:
1- it consumed even more memory than the first strategy.
2- it prolonged installation time

Third strategy: Profile Guided Compilation

This strategy was used in Android 7 (Nougat) and Android 8 (Oreo).
First it compiled using JIT.



Since there were portions of code that needed to execute rapidly, ART could compile and cache them.



According to the Profile Guided Compilation strategy, the rapid code (named “profile data”) is stored in the OAT (Optimized Android Application Package) after being compiled. Meanwhile, the rest of the code remains uncompiled until the user needs it. This, fortunately, had no negative effect on installation time.
But there remained one small problem :
How do we find the profile data?

Fourth Strategy : Profile in the Cloud

This current strategy has been used since Android 9 (pie).
The main idea in this strategy is that most users use the app the same way; so we can get the profile data from the app’s first users.
When new users install the app, they are installing apk along with an additional file called core profile (the profile data from previous users).