Gradle for Android

(a four-hour tour)

Segment #1: Introducing Gradle for Android

What Is Gradle?

- Role: Build Automation
  - Think Ant plus Maven plus other goodness
- Implementation: It's Groovy
  - DSL implemented in Groovy, blending declarative structures and full-blown scripting
- Provider: Gradleware
  - Open source, Apache licensed

What Is Gradle for Android?

- Gradle plugin
- Enables Gradle scripts to build Android apps
  - Compilation and cross-compilation
  - Resource and asset packaging
  - APK assembly and signing
- Written by Google

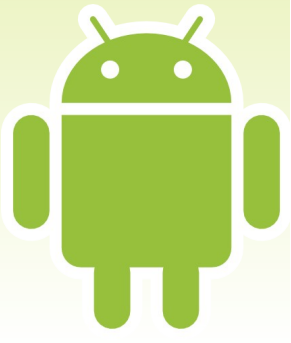
Why Is Google Doing This?

- One Build System (To Rule Them All)
  - Android Studio
  - Command-line builds / CI servers
  - Other IDEs?
- Polyglot Build System (Java, C/C++, etc.)
- Malleable
  - Easier to extend with less breakage, compared to original Ant tasks

Gotta Getta Gradle

- Direct Download
  - `http://www.gradle.org/downloads`
  - Gradle 2.2.1
    - Used by Android Studio 1.1 and the Gradle for Android plugin 1.1

`http://tools.android.com/tech-docs/new-build-system/version-compatibility`



Gotta Getta Gradle

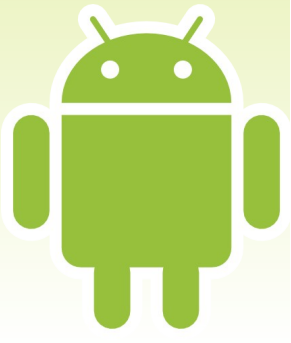
- The Gradle Wrapper: Pieces
  - **`gradlew`** script and batch file
  - `gradle/wrapper/` directory
    - JAR file
    - `gradle-wrapper.properties`

Gotta Getta Gradle

- The Gradle Wrapper: Objectives
  - Record the required Gradle version
    - distributionUrl property
  - Command-line bootstrap

Gotta Getta Gradle

- Android Studio
  - New Projects
    - Sets up Gradle wrapper for a compatible version of Gradle
    - Exact version based on Gradle for Android plugin version
  - Imported Projects
    - Needs `gradle-wrapper.properties` file or needs a standalone copy of Gradle installed



Gotta Getta Gradle

- The Gradle Wrapper: Rules
  - Validate Your `distributionUrl`
    - Make sure it points to `gradle.org` or other trusted source
  - Only use **`gradlew`** from a trusted source
    - Not used by Android Studio, so only relevant if you are doing command-line builds
- Rules Rationale
  - Scripts, JAR, and Gradle run on your development machine!

The Basic Gradle Process

- Write `build.gradle` File
  - Describes sources and results
  - Same role as build.xml for Ant, etc.
  - Usually in root of project directory
- Run **`gradle`** / **`gradlew`**
  - Supply task name as command-line parameter
  - Optional: IDE integration

Escape From Eclipse

- Migrating to Android Studio
  - Run Android Studio Eclipse import wizard
  - Results
    - Copy made of project
    - Copy reorganized into new project structure
    - Gradle build files created
    - Report generated of changes made

Escape From Eclipse

- Exporting a `build.gradle`
  - Export wizard in Eclipse through current ADT
  - Out of date
- Manually Assembling a `build.gradle`
  - Standard “template” for an Eclipse-style project that can be built by Gradle for Android

Escape from Eclipse

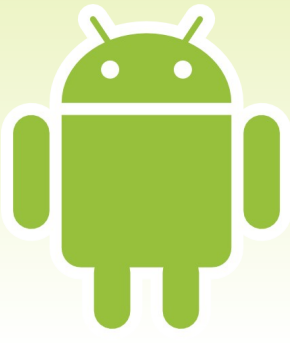
- Just Go With Android Studio
  - New-project wizard creates project with app module
  - Two `build.gradle` files
    - Project-level one: stuff for all modules
    - Module-level one: stuff for this module only
    - Net for most apps: the difference is academic, as you will only have one module
  - `settings.gradle` file
    - Indicates the roster of modules for this project

build.gradle: High-Level View

- `buildscript {}`
  - Describing dependencies for running the build
  - Key: Android plugin
- `apply plugin: 'com.android.application'`
- `dependencies {}`
  - Describing compile-time dependencies (JARs, etc.)
- `android {}`
  - Tailoring what Android builds for you

Project Configuration Options

- `versionCode`
- `versionName`
- `minSdkVersion`
- `targetSdkVersion`
- `applicationId`
- `signingConfig`



Tons o' Tasks

- `assemble*`
  - Compiles APK for you
  - Tied to "build type" (`assembleDebug`, `assembleRelease` are default)
- `install*`
  - Installs APK on device for you, after assembly
  - Only `installDebug` works by default
    - `installRelease` requires configuring your signing keys

Project Structures, Old and New

- Original Recipe
  - `src/`, `res/`, `assets/` in top-level project directory
  - `libs/` also in top-level project directory
- New Project Structure
  - `java/`, `res/`, `assets/` in subdirectory
    - `main/` by default
    - Others by “build type” or “product flavor”
  - `libs/` remains in top-level directory
    - Or gone, replaced by artifacts

Segment #2: Source Sets, Build Types, and Product Flavors

Pieces of New Project Structure

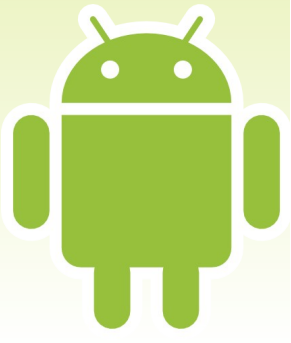
- Source Sets
  - Ummmm... sets of source 'n stuff!
- Build Types
  - Stages of your development lifecycle
- Product Flavors
  - Different outputs for different audiences
- Build Variants
  - Cross product of build types and product flavors

Source Sets

- Gradle Construct for Organizing “Source”
  - In Android’s case, includes resources and assets
- Vision
  - Have one `main/` source set with most of your code
  - Have alternatives in other source sets, used conditionally
    - Resources, assets: can replace `main/` source set
    - Java: cannot replace `main/`, can only add

Build Types

- Android Plugin Construct for Describing Output Variations
  - Two build types come default: `debug` and `release`
- Build Types Configurable
  - Project properties in `build.gradle`
  - Source sets
- Define Others As Needed
  - Smoke tests, “dogfooding” builds, etc.

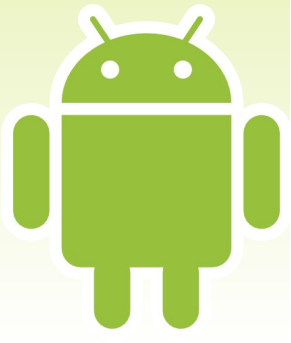


Build Types & Source Sets

- Create a source set with same name as build type
  - E.g., `src/debug/`
- Supply your changes
  - Resources
  - Assets
  - Manifest
- If have one source set for every build type, can also supply N implementations of Java classes

Build Types & build.gradle

- Key Configurable Properties
  - `applicationIdSuffix`
  - `versionNameSuffix`
  - `signingConfig`
  - `debuggable`
  - `minifyEnabled`
  - `buildConfigField`



Defining Custom Build Types

- Step #1: Define the name and initialize from some other build type
- Step #2: Configure as desired
- Step #3: Beer!

Product Flavors and Build Variants

- Product Flavors
  - Android plugin construct for different deployment variations
  - None defined by default, can create your own
- Build Variants
  - Cross product of build types and product flavors
  - Drive task names (`assembleAmazonDebug`) and results

Defining Product Flavors

- Declare in `android` closure
- Define desired properties
- Add source sets
- Limitation: build type, product flavor names must be unique

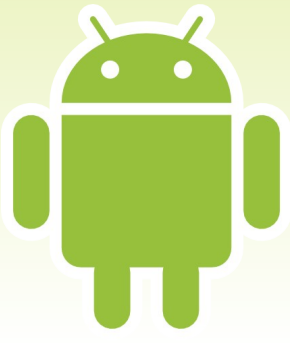
```
productFlavors {
    flavor1 {  ...  }
    flavor2 {  ...  }
}
```

Product Flavor Configuration

- Source Sets
  - Same rules as for build types
- `build.gradle` Options
  - Everything that you can define for the main project itself
  - Inherits from `defaultConfig`
  - Build type usually overlays product flavor
    - E.g., package name

Product Flavor Scenarios

- Ecosystem
  - Different source for Google's IAP versus Amazon's
- Other Play Store Splits
  - OpenGL texture compressions
  - Screen sizes
  - Platform versions
  - Etc.



Splits

- Opt-In “Automatic” Product Flavors
  - CPU architecture (x86 vs. ARM)
  - Screen density

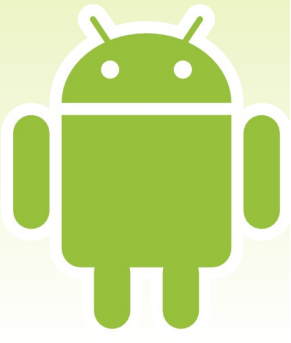
```
splits {
    abi {
      enable true
      reset()
      include 'x86', 'armeabi-v7a', 'mips'
      universalApk true
    }
  }
```

Manifest Merging

- Prioritization
  - `build.gradle`
  - Build types and product flavors
  - The `main` sourceset
  - Manifests contributed by library projects / AARs

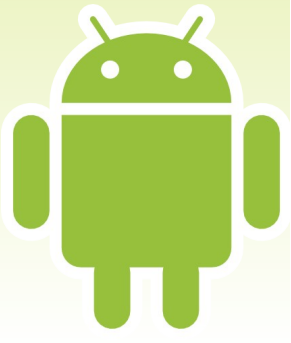
Manifest Merging

- Default Merger Rules
  - Disparate elements are merged
  - Identical elements have attributes merged
    - …unless there is an identical attribute with different values, which results in a build error
- Overrides
  - `tools:node` for elements
  - `tools:replace` and `tools:remove` for attributes

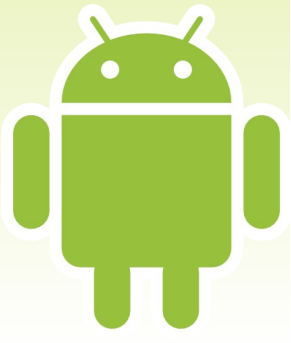


Revisiting the Legacy Gradle File

- Default Settings = New Project Structure
- Map Other Directories Into `main` Source Set
  - `manifest.srcFile`
  - `java.srcDirs`
  - Etc.
- Set Roots for Build Types, Product Flavors
  - Theoretically allows for build types and product flavors alongside legacy structure



Segment #3: Dependencies



Dependency Scopes

- In `buildscript`
  - Controls the build process itself
  - Key one: Gradle for Android plugin
- Outside of `buildscript`
  - Dependencies for the app being built

Depending Upon a Local JAR

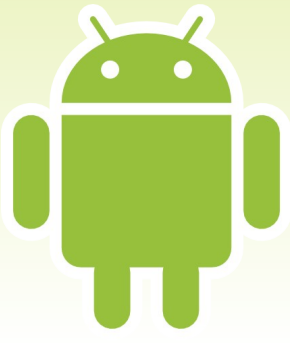
- Official Answer: Just Say No!
  - Replace with artifacts if at all possible
- Official Grudging Support
  - Put it in `libs/`
  - Add compile `fileTree()` statement to pull those in as dependencies

Depending Upon a JAR Artifact

- What's an Artifact
  - Entry in an artifact repository
  - JAR (or AAR) plus metadata
  - Predominant repository type: Maven
  - Analog of Ruby "gems", Bower for JavaScript, etc.

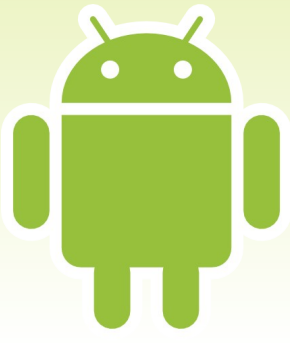
Depending Upon a JAR Artifact

- Where Are Artifact Repositories?
  - Public and popular: Maven Central, JCenter
  - Public: Own hosted repository
    - CWAC libraries
  - Private
    - Enterprise development
  - Local
    - Android Repository, Google Repository



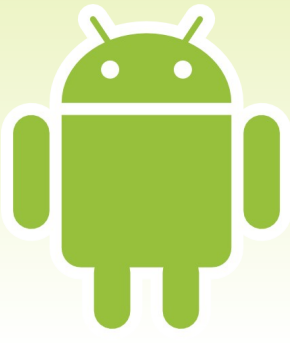
Android Support Package

- Some of the Available Artifacts
  - `support-v4` & `support-v13`
    - Use only one of these!
  - `appcompat-v7`
  - `gridlayout-v7`
  - `mediarouter-v7`
  - `recyclerview-v7`
  - `cardview-v7`
  - `leanback-v17`



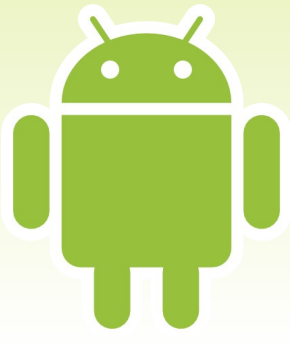
Depending Upon a JAR Artifact

- Step #1: Identify Your Artifact and Repository
  - Example: Android Arsenal
- Step #2: Add Repository to `build.gradle`
  - If not there already from previous work
- Step #3: Add artifact via compile statement to `build.gradle`
  - `compile 'com.commonsware.cwac:richedit:0.5.+'`



Gradle and the Android Library Project

- Creating a Library Project
  - Use `com.android.library` plugin instead of `com.android.application` plugin
- Consuming Your Own Library Project
  - Option #1: Set it up as a sub-project
  - Option #2: Publish it as an AAR and use that
- Consuming a Third-Party Library Project
  - Option #1: Use the AAR
  - Option #2: Set it up as a sub-project



Depending Upon Another Project

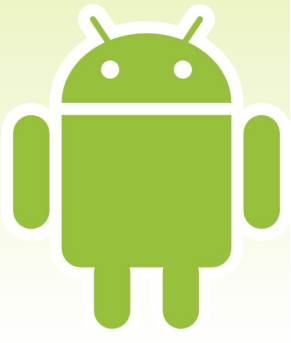
- `settings.gradle`
  - Include statement identifying all projects that are part of the app
  - Leading colon indicates a directory under the overall root
  - Other colons used as path separators

```
include ':HelloLibraryConsumer', ':libraries:HelloLibrary'
```

Depending Upon Another Project

- `build.gradle`
  - Add `compile project()` directive to `dependencies` closure to point to other project's code to include in your own

```
compile project(':libraries:HelloLibrary')
```



Enter the AAR

- Packaged Library Project
  - *Compiled* code
    - Contrast with standard library project, which normally requires full source
  - Resources, assets, manifest entries
- Artifact, Designed for Repositories
  - Promote reuse
  - Many open source libraries publish AARs today

Depending Upon AARs

- AARs in Repositories
  - Identical to depending upon JARs
  - May need to add @aar suffix
    - Only if JAR and AAR both published, not common
- AARs Outside of Repositories
  - Not officially supported
  - Various recipes floating around

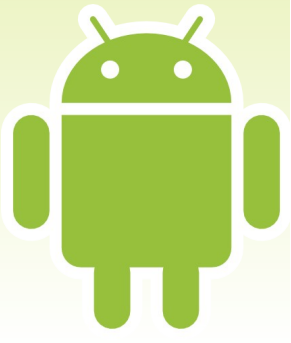
Publishing an AAR

- Maven Plugin
  - Configure your POM in an `uploadArchives` task
  - Configure Repository
    - mavenDeployer
    - SSH/SCP
    - Directly-accessible directory (`file://`)

Conditional Dependencies

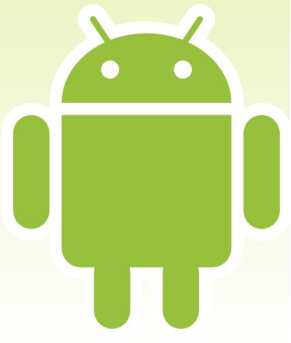
- Statements in `dependencies` imply where they are used
  - `compile` = use in all builds
  - `androidTestCompile` = use in instrumentation tests
  - `debugCompile` = use in `debug` build type
    - Example: Robotium
  - `flavor1Compile` = use in `flavor1` product flavor
    - Example: Play Services SDK

Segment #4: More Tips and Techniques



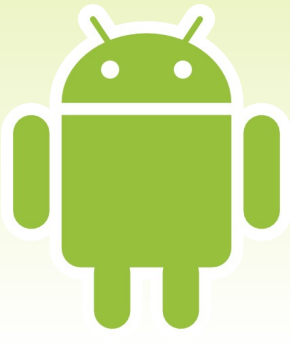
Testing with Gradle: Apps

- Set up `androidTest` source set in app's own project
  - Code will be used for testing, not for standard APK builds
  - No need for manifest
- Put your instrumentation test code in that source set
  - JUnit3
  - JUnit4
  - UI Automator 2.0



Testing with Gradle: Apps

- `defaultConfig`
  - `testApplicationId`
  - `testInstrumentationRunner`
    - Classically `android.test.InstrumentationTestRunner`
- `gradle connectedCheck`
  - Runs tests in parallel across all connected devices and emulators
  - Installs app and tests, runs tests, uninstalls
- Output in `build/reports/androidTests/connected`



Testing with Gradle: Libraries

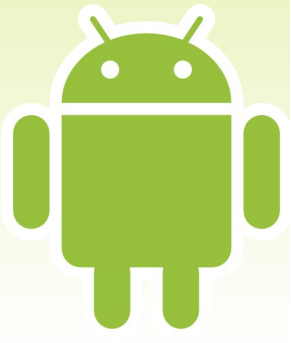
- Works Just the Same!
  - Add `androidTest` source set
  - Put tests in there
  - Add `defaultConfig` settings
  - Run `connectedCheck` task
- Tests + library code → APK for testing

Defining Custom Tasks

- Standalone Tasks
  - Ones not tied to any build variant
  - Just use `task` keyword

```
task hello {
    doLast {
        println 'Hello world!'
    }
}

task hello << {
    println 'Hello world!'
}
```



Defining Custom Tasks

- Per-Variant Tasks
  - Iterate over variants
    - android.applicationVariants or android.libraryVariants
  - Create new task with dynamic name
  - Configure new task, including dependencies

Gradle Plugins

- Tie into Gradle, Gradle for Android
- Canned solutions for problems
  - Versus having to copy or include common Gradle scripts
- A bit challenging, given limited Gradle for Android DSL documentation
- Example: Trello and Victor
  - http://blog.danlew.net/2015/04/27/victor/

Properties Files

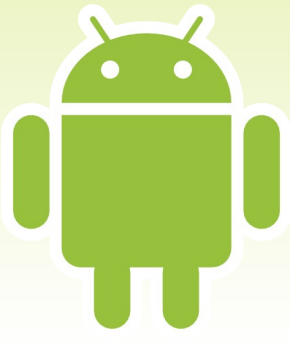
- `gradle.properties`
  - Reside in user's home directory (e.g., `~/.gradle`)
  - Contents automatically exposed as "globals"
  - Read-only
- Custom Properties Files
  - Access java.util.Properties and java.io.FileInputStream from Gradle
  - Do what you want!

Environment Variables

- Option #1: Prefixed
  - `ORG_GRADLE_PROJECT_foo` shows up as `foo` in scripts
- Option #2: Arbitrary
  - `System.getenv('foo')`
  - `System.env.foo`
- WARNING: Only for command-line!
  - Android Studio does not pass environment variables along to its `gradle` child process

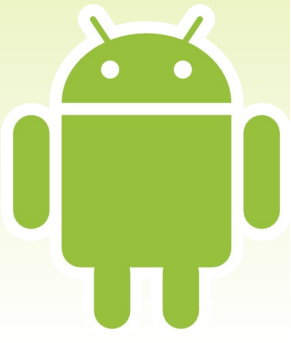
NDK and Gradle

- Option #1: Externally-Built Binaries
  - Example: SQLCipher for Android
  - Add `jniLibs/` to a source set, with standard architecture directories underneath
    - e.g., `jniLibs/armeabi-v7a/`



NDK and Gradle

- Option #2: Makefile-Free Builds
  - `local.properties` with `ndk.dir` pointing to your NDK installation
  - ndk closure in `defaultConfig`
    - `moduleName`
    - `cFlags`
    - `ldLibs`
  - ndk closures in product flavor definitions
    - `abiFilter`



NDK and Gradle

- Option #3: Execute `ndk-build` from Custom Task
  - `https://gist.github.com/dweinstein/7528167`
  - Includes executing `ndk-build`, packaging binaries for the right builds
  - Benefit: uses your existing makefiles
    - Good for makefiles that do more than the norm
  - Cost: unofficial, YMMV

Other Types of Source

- AIDL
  - Put in `aidl/` directory within source sets
  - Or, configure `aidl.srcDirs`
- Renderscript
  - Put in `rs/` directory within source sets
  - Or, configure `renderscript.srcDirs`
  - Add `renderscriptTargetApi` and optionally `renderscriptSupportMode`

Where To Learn More

- `http://tools.android.com/`
  - Home of the Android tools team
  - Information on Gradle for Android, Android Studio
    - Note: much is out of date!
- `http://gradle.org`
  - For general Gradle information
- `http://commonsware.com/Android`
  - Some book by some balding guy
  - Several chapters on Gradle for Android