LEARNING gradle

#gradle

Table of Contents

About

Chapter 1: Getting started with gradle

Remarks

Gradle is an open-source, general-purpose build tool. It is popular in the Java community and is the preferred build tool for Android.

Highlighted Gradle features

- Declarative build scripts *are* code written in Groovy or Kotlin.
- Lots of core and community plugins which use a flexible, convention-based approach
- Incremental builds such that tasks whose dependencies who haven't changed aren't rerun.
- Built-in dependency resolution for Maven and Ivy. Contributed plugins provide dependency resolution from other `repositories` such as npm.
- First-class multi-project builds.
- Integration with other build tools like Maven, Ant and others.
- Build Scans that increase developers' the ability to collaborate on and optimize Gradle builds.

More information

If you want to learn more about Gradle features can look at the Overview part of the Gradle User Guide.

If you want to try Gradle can check out the guides here. You can walk through a Java quickstart guide, learn how use Gradle for the first time, and migrate from another build tool.

Examples

Gradle Installation

Requirements: Installed Java JDK or JRE (version 7 or higher for Gradle 3.x version)

Installation steps:

1. Download Gradle distribution from the official web site
2. Unpack the ZIP
3. Add the `GRADLE_HOME` environment variable. This variable should point to the unpacked files from the previous step.
4. Add `GRADLE_HOME/bin` to your `PATH` environment variable, so you can run Gradle from the command line interface (CLI)
5. Test your Gradle installation by typing `gradle -v` in the CLI. The output should contain the installed Gradle version and the current Gradle configuration details

More information can be found in the official user guide

Installation with homebrew on OS X / macOS

Users of homebrew can install gradle by running

```
brew install gradle
```

Installing with SdkMan

Users of SdkMan can install Gradle by running:

```
sdk install gradle
```

Install specific version

```
sdk list gradle
sdk install gradle 2.14
```

Switch versions

```
sdk use gradle 2.12
```

Install Gradle plugin for Eclipse

Here are the steps required to install Gradle plugin in Eclipse:

1. Open Eclipse and go to **Help** -> **Eclipse Marketplace**
2. In the search bar, enter **buildship** and hit enter
3. Select **"Buildship Gradle Integration 1.0"** and click **Install**
4. In the next window, click **Confirm**
5. Then, **accept** the terms and license of agreement, then click **Finish**
6. After installation, Eclipse will need to restart, click **Yes**

Hello World

Gradle tasks can be written using Groovy code from inside a projects build.gradle file. These tasks can then be executed using `> gradle [taskname]` at the terminal or by executing the task from within an IDE such as Eclipse.

To create the Hello World example in gradle we must define a task that will print a string to the console using Groovy. We will use Groovy's `printLn` to call Java's `System.out.printLn` method to print the text to the console.

build.gradle

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

We can then execute this task by using `> gradle hello` or `> gradle -q hello`. The `-q` is used to suppress gradle log messages so that only the output of the task will be shown.

Output of `> gradle -q hello`:

```
> gradle -q hello
Hello world!
```

More about tasks

First of all: operator `<<` (leftShift) is equivalent of `doLast {closure}`. From **gradle 3.2** it is **deprecated**. All the task code are writing in a **build.gradle**.

> A task represents some atomic piece of work which a build performs. This might be compiling some classes, creating a JAR, generating Javadoc, or publishing some archives to a repository.

Gradle supports two big types of tasks: simple and enhanced.

Let's observe some task definition styles:

```
task hello {
    doLast{
       //some code
    }
}
```

Or the:

```
task(hello) {
    doLast{
       //some code
    }
}
```

This tasks above are equivalents. Also, you can provide some extensions to the task, such as: `dependsOn`,`mustRunAfter`, `type` etc. You can extend task by adding actions after task definition, like this:

```
task hello {
    doLast{
       println 'Inside task'
    }
}
hello.doLast {
    println 'added code'
}
```

When we'll execute this we got:

```
> gradle -q hello
    Inside task
    added code
```

Questions about task dependencies and ordering examined here

Let's talk about two big types of task.

Simple:

Tasks which we define with an action closure:

```
    task hello {
        doLast{
        println "Hello from a simple task"
        }
    }
```

Enhanced

Enhanced it is a task with a preconfigured behavior. All plugins that you using in your project are the *extended*, or the **enhanced tasks**. Let's create ours and you will understand how it works:

```
task hello(type: HelloTask)

class HelloTask extends DefaultTask {
    @TaskAction
    def greet() {
        println 'hello from our custom task'
    }
}
```

Also, we can pass parameters to our task, like this:

```
class HelloTask extends DefaultTask {
    String greeting = "This is default greeting"
    @TaskAction
    def greet() {
        println greeting
    }
}
```

And from now on we can rewrite our task like so:

```
   //this is our old task definition style
task oldHello(type: HelloTask)
   //this is our new task definition style
task newHello(type: HelloTask) {
    greeting = 'This is not default greeting!'
}
```

When we'll execute this we got:

```
> gradle -q oldHello
This is default greeting

> gradle -q newHello
This is not default greeting!
```

All questions about development gradle plugins onto official site

Read Getting started with gradle online: https://riptutorial.com/gradle/topic/894/getting-started-with-gradle

Chapter 2: Auto Increment Version Number Using Gradle Script For Android Applications

Examples

How To Call Auto Increment Method When Build

```
gradle.taskGraph.whenReady {taskGraph ->
    if (taskGraph.hasTask(assembleDebug)) {  /* when run debug task */
        autoIncrementBuildNumber()
    } else if (taskGraph.hasTask(assembleRelease)) { /* when run release task */
        autoIncrementBuildNumber()
    }
}
```

Auto Increment Method Definition

```
 /*Wrapping inside a method avoids auto incrementing on every gradle task run. Now it runs
only when we build apk*/
ext.autoIncrementBuildNumber = {

    if (versionPropsFile.canRead()) {
        def Properties versionProps = new Properties()
        versionProps.load(new FileInputStream(versionPropsFile))
        versionBuild = versionProps['VERSION_BUILD'].toInteger() + 1
        versionProps['VERSION_BUILD'] = versionBuild.toString()
        versionProps.store(versionPropsFile.newWriter(), null)
    } else {
        throw new GradleException("Could not read version.properties!")
    }
}
```

Read and Assign Version Number from a property file to a variable

def versionPropsFile = file('version.properties') def versionBuild

```
/*Setting default value for versionBuild which is the last incremented value stored in the
file */
if (versionPropsFile.canRead()) {
    def Properties versionProps = new Properties()
    versionProps.load(new FileInputStream(versionPropsFile))
    versionBuild = versionProps['VERSION_BUILD'].toInteger()
} else {
    throw new GradleException("Could not read version.properties!")
}
```

Read Auto Increment Version Number Using Gradle Script For Android Applications online: https://riptutorial.com/gradle/topic/10696/auto-increment-version-number-using-gradle-script-for-android-applications

Chapter 3: Dependencies

Examples

Add a Local JAR File Dependency

Single JAR

Sometimes you have a local JAR file you need to add as a dependency to your Gradle build. Here's how you can do this:

```
dependencies {
    compile files('path/local_dependency.jar')
}
```

Where `path` is a directory path on your filesystem and `local_dependency.jar` is the name of your local JAR file. The `path` can be relative to the build file.

Directory of JARs

It's also possible to add a directory of jars to compile. This can be done like so:

```
dependencies {
        compile fileTree(dir: 'libs', include: '*.jar')
}
```

Where `libs` would be the directory containing the jars and `*.jar` would be the filter of which files to include.

Directory of JARs as repository

If you only want to lookup jars in a repository instead of directly adding them as a dependency with their path you can use a flatDir repository.

```
repositories {
    flatDir {
        dirs 'libs'
    }
}
```

Looks for jars in the *`libs`* directory and its child directories.

Add a Dependency

Dependencies in Gradle follow the same format as Maven. Dependencies are structured as follows:

```
group:name:version
```

Here's an example:

```
'org.springframework:spring-core:4.3.1.RELEASE'
```

To add as a compile-time dependency, simply add this line in your `dependency` block in the Gradle build file:

```
compile 'org.springframework:spring-core:4.3.1.RELEASE'
```

An alternative syntax for this names each component of the dependency explicitly, like so:

```
compile group: 'org.springframework', name: 'spring-core', version: '4.3.1.RELEASE'
```

This adds a dependency at compile time.

You can also add dependencies only for tests. Here's an example:

```
testCompile group: 'junit', name: 'junit', version: '4.+'
```

Depend on Another Gradle Project

In the case of a multi-project gradle build, you may sometimes need to depend on another project in your build. To accomplish this, you'd enter the following in your project's dependencies:

```
dependencies {
    compile project(':OtherProject')
}
```

Where `':OtherProject'` is the gradle path for the project, referenced from the root of the directory structure.

To make `':OtherProject'` available in the context of the `build.gradle` file add this to the corresponding `settings.gradle`

```
include ':Dependency'
project(':Dependency').projectDir = new File('/path/to/dependency')
```

For a more detailed explanation, you can reference Gradle's official documentation here.

List Dependencies

Calling the `dependencies` task allows you to see the dependencies of the root project:

```
gradle dependencies
```

The results are dependency graphs (taking into account transitive dependencies), broken down by configuration. To restrict the displayed configurations, you can pass the `--configuration` option followed by one chosen configuration to analyse:

```
gradle dependencies --configuration compile
```

To display dependencies of a subproject, use `<subproject>:dependencies` task. For example to list dependencies of a subproject named `api`:

```
gradle api:dependencies
```

Adding repositories

You have to point Gradle to the location of your plugins so Gradle can find them. Do this by adding a `repositories { ... }` to your `build.gradle`.

Here's an example of adding three repositories, JCenter, Maven Repository, and a custom repository that offers dependencies in Maven style.

```
repositories {
  // Adding these two repositories via method calls is made possible by Gradle's Java plugin
  jcenter()
  mavenCentral()

  maven { url "http://repository.of/dependency" }
}
```

Add .aar file to Android project using gradle

1. Navigate to project's `app` module and create `libs` directory.
2. Place your `.aar` file there. For example `myLib.aar`.
3. Add the code below to `android` block of `app` level's `build.gradle` file.

```
  repositories {
        flatDir {
            dirs 'libs'
        }
    }
```

This way you defined a new extra repository that points to `app` module's `libs` folder.

4. Add the code below to `dependencies` block or the `build.gradle` file:

```
compile(name:'myLib', ext:'aar')
```

Read Dependencies online: https://riptutorial.com/gradle/topic/2524/dependencies

Chapter 4: Gradle Init Scripts

Examples

Add default repository for all projects

Add a init.gradle to your user gradle folder. The init.gradle is recognized on every project.

```
Unix: ~/.gradle/init.gradle
```

> These are also alternative locations where init script can be placed and loaded automatically:-
>
> - Any *.**gradle** file in **USER_HOME/.gradle/init.d**
> - Any *.**gradle** file in the Gradle installation's **init.d** directory

init.gradle with mavenLocal as repository in all projects.

```
allprojects {
    repositories {
        mavenLocal()
    }
}
```

With that you have your local maven cache available in all repositories. A use case could be to use a jar that you put in ther with "gradle install" in another project without adding the mavenLocal repository to the build.gradle or adding a nexus/artifactory server.

Read Gradle Init Scripts online: https://riptutorial.com/gradle/topic/4234/gradle-init-scripts

Chapter 5: Gradle Performance

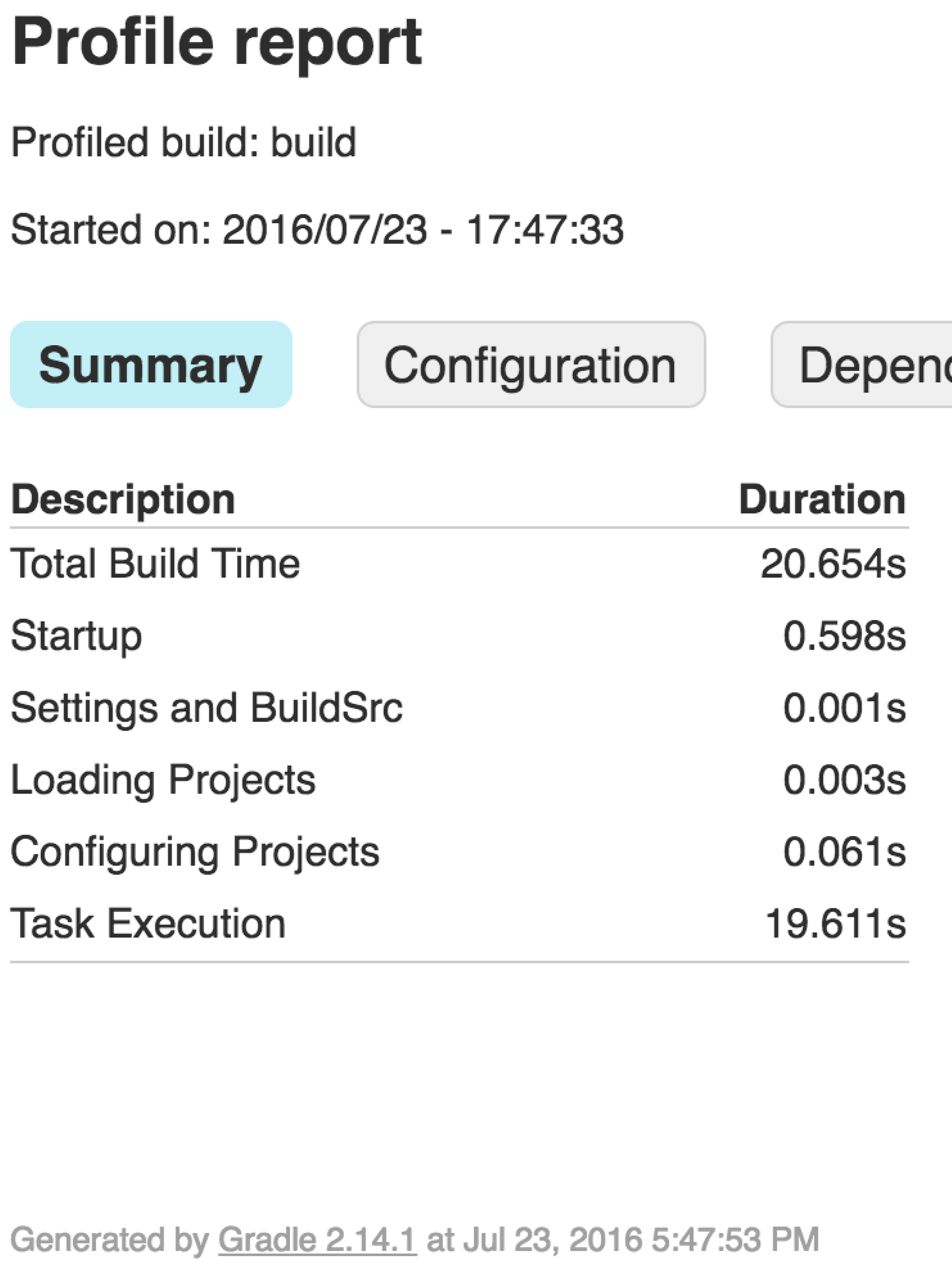
Examples

Profiling a Build

Before you begin tuning your Gradle build for performance, you should establish a baseline and figure out which portions of the build are taking the most time. To do this, you can profile your build by adding the `--profile` argument to your Gradle command:

```
gradle --profile
./gradlew --profile
```

After the build is complete, you will see an HTML profile report for the build under `./build/reports/profile/`, looking something like this:



Profile report

Profiled build: build

Started on: 2016/07/23 - 17:47:33

Description	Duration
Total Build Time	20.654s
Startup	0.598s
Settings and BuildSrc	0.001s
Loading Projects	0.003s
Configuring Projects	0.061s
Task Execution	19.611s

Generated by Gradle 2.14.1 at Jul 23, 2016 5:47:53 PM

By clicking on the tabs next to **Summary**

, you can see a more-detailed breakdown of where time is spent.

Configure on Demand

If profiling your build shows significant time spend in **Configuring Projects**, the Configure on Demand option might improve your performance.

You can enable Configure on Demand mode by editing `$GRADLE_USER_HOME/.gradle/gradle.properties` (`~/.gradle/gradle.properties` by default), and setting `org.gradle.configureondemand`.

```
org.gradle.configureondemand=true
```

To enable it only for a specific project, edit that project's `gradle.properties` file instead.

If Configure on Demand is enabled, instead of configuring all projects up front, Gradle will only configure projects that are needed for the task being run.

From the Gralde Manual:

> Configuration on demand mode attempts to configure only projects that are relevant for requested tasks, i.e. it only executes the `build.gradle` file of projects that are participating in the build. This way, the configuration time of a large multi-project build can be reduced. In the long term, this mode will become the default mode, possibly the only mode for Gradle build execution.

Tuning JVM Memory Usage Parameters for Gradle

You can set or increase memory usage limits (or other JVM arguments) used for Gradle builds and the Gradle Daemon by editing `$GRADLE_USER_HOME/.gradle/gradle.properties` (`~/.gradle/gradle.properties` by default), and setting `org.gradle.jvmargs`.

To configure these limits only for a specific project, edit that project's `gradle.properties` file instead.

The default memory usage settings for Gradle builds and the Gradle Daemon are:

```
org.gradle.jvmargs=-Xmx1024m -XX:MaxPermSize=256m
```

This allows a general maximum memory allocation (heap size) of 1GB, and a maximum memory allocation for permanent "internal" objects of 256MB. When these sizes are reached, Garbage Collection occurs, which can decrease performance significantly.

Assuming you have the memory to spare, you could easily double these like so:

```
org.gradle.jvmargs=-Xmx2024m -XX:MaxPermSize=512m
```

Note that you'll stop seeing benefit from increasing `XX:MaxPermSize` sooner than when `Xmx` increases stop becoming beneficial.

Use the Gradle Daemon

You can enable the Gradle Daemon to improve the performance of your builds.

The Gradle Daemon keeps the Gradle Framework initialized and running, and caches project data in memory to improve performance.

For a Single Build

To enable the Daemon for a single build, you can simply pass the `--daemon` argument to your `gradle` command or Gradle Wrapper script.

```
gradle --daemon
./gradlew --daemon
```

For All Builds of a Project

To enable the Daemon for all builds of a project, you can add:

```
org.gradle.daemon=true
```

To your project's `gradle.properties` file.

For All Builds

To enable the Gradle Daemon by default, for every build made by your user account on your system, edit `$GRADLE_USER_HOME/.gradle/gradle.properties` (`~/.gradle/gradle.properties` by default) and add this line:

```
org.gradle.daemon=true
```

You can also do this in a single command on Mac/Linux/*nix systems:

```
touch ~/.gradle/gradle.properties && echo "org.gradle.daemon=true" >>
~/.gradle/gradle.properties
```

Or on Windows:

```
(if not exist "%USERPROFILE%/.gradle" mkdir "%USERPROFILE%/.gradle") && (echo
org.gradle.daemon=true >> "%USERPROFILE%/.gradle/gradle.properties")
```

Disabling the Daemon

You can disable the Daemon for a specific build using the `--no-daemon` argument, or disable it for a specific project by explicitly setting `org.gradle.daemon=false` in the project's `gradle.properties` file.

Stopping the Daemon

If you wish to stop a Daemon process manually, you can either kill the process via your operating system task manager or run the `gradle --stop` command. The `--stop` switch causes Gradle to request that all running Daemon processes, of the same Gradle version used to run the command, terminate themselves. Ordinarily, Daemon processes will automatically terminate themselves *after *3 hours of inactivity or less*.

Gradle Parallel builds

Gradle will only run one task at a time by default, regardless of the project structure. By using the `--parallel` switch, you can force Gradle to execute independent subprojects - those that have no implicit or explicit project dependencies between one another - in parallel, allowing it to run multiple tasks at the same time as long as those tasks are in different projects.

To build a projects in parallel mode:

```
gradle build --parallel
```

You can also make building in parallel the default for a project by adding the following setting to the project's gradle.properties file:

```
org.gradle.parallel=true
```

Use latest Gradle version

The Gradle team works regularly on improving the performance of different aspects of Gradle builds. If you're using an old version of Gradle, you're missing out on the benefits of that work. Try upgrading to the latest version of Gradle to see what kind of impact it has. Doing so is low risk because very few things break between minor versions of Gradle.

The properties file for the Gradle wrapper can be found in your project folder under `gradle/wrapper/` and is called `gradle-wrapper.properties`. The content of that file might look like this:

```
distributionBase=GRADLE_USER_HOME
distributionPath=wrapper/dists
zipStoreBase=GRADLE_USER_HOME
zipStorePath=wrapper/dists
distributionUrl=https\://services.gradle.org/distributions/gradle-X.X.X.zip
```

You can manually change the version number `X.X.X`(current version) to `Y.Y.Y`(newer version) and the next time you run the wrapper, the new version is downloaded automatically.

Read Gradle Performance online: https://riptutorial.com/gradle/topic/3443/gradle-performance

Chapter 6: Gradle Plugins

Examples

Simple gradle plugin from `buildSrc`

Simple example of how to create a custom plugin and DSL for your gradle project.
This sample uses one of the three possible ways of creating plugins.
The three ways are:

- inline
- buildSrc
- standalone plugins

This example shows creating a plugin from the **buildSrc** folder.

This sample will create five files

```
// project's build.gradle
build.gradle
// build.gradle to build the `buildSrc` module
buildSrc/build.gradle
// file name will be the plugin name used in the `apply plugin: $name`
// where name would be `sample` in this example
buildSrc/src/main/resources/META-INF/gradle-plugins/sample.properties
// our DSL (Domain Specific Language) model
buildSrc/src/main/groovy/so/docs/gradle/plugin/SampleModel.groovy
// our actual plugin that will read the values from the DSL
buildSrc/src/main/groovy/so/docs/gradle/plugin/SamplePlugin.groovy
```

build.gradle:

```
group 'so.docs.gradle'
version '1.0-SNAPSHOT'

apply plugin: 'groovy'
// apply our plugin... calls SamplePlugin#apply(Project)
apply plugin: 'sample'

repositories {
    mavenCentral()
}

dependencies {
    compile localGroovy()
}

// caller populates the extension model applied above
sample {
    product = 'abc'
    customer = 'zyx'
}
```

```
// dummy task to limit console output for example
task doNothing <<{}
```

buildSrc/build.gradle

```
apply plugin: 'groovy'

repositories {
    mavenCentral()
}

dependencies {
    compile localGroovy()
}
```

buildSrc/src/main/groovy/so/docs/gradle/plugin/SamplePlugin.groovy:

```
package so.docs.gradle.plugin

import org.gradle.api.Plugin
import org.gradle.api.Project

class SamplePlugin implements Plugin<Project> {
    @Override
    void apply(Project target) {
        // create our extension on the project for our model
        target.extensions.create('sample', SampleModel)
        // once the script has been evaluated the values are available
        target.afterEvaluate {
            // here we can do whatever we need to with our values
            println "populated model: $target.extensions.sample"
        }
    }
}
```

buildSrc/src/main/groovy/so/docs/gradle/plugin/SampleModel.groovy:

```
package so.docs.gradle.plugin

// define our DSL model
class SampleModel {
    public String product;
    public String customer;

    @Override
    public String toString() {
        final StringBuilder sb = new StringBuilder("SampleModel{");
        sb.append("product='").append(product).append('\'');
        sb.append(", customer='").append(customer).append('\'');
        sb.append('}');
        return sb.toString();
    }
}
```

buildSrc/src/main/resources/META-INF/gradle-plugins/sample.properties

```
implementation-class=so.docs.gradle.plugin.SamplePlugin
```

Using this setup we can see the values supplied by the caller in your DSL block

```
 $ ./gradlew -q doNothing
SampleModel{product='abc', customer='zyx'}
```

How to write a standalone plugin

To create a custom standalone Gradle plug-in using java (you can also use Groovy) you have to create a structure like this:

```
plugin
|-- build.gradle
|-- settings.gradle
|-- src
    |-- main
    |   |-- java
    |   |-- resources
    |       |-- META-INF
    |           |-- gradle-plugins
    |-- test
```

Setup gradle configuration

In the `build.gradle` file you define your project.

```
apply plugin: 'java'
apply plugin: 'maven'

dependencies {
    compile gradleApi()
}
```

The `java` plugin will be used to write java code.
The `gradleApi()` dependency will give us all method and propertiess needed to create a Gradle plugin.

In the `settings.gradle` file:

```
rootProject.name = 'myplugin'
```

It will define the **artifact id** in Maven.
If `settings.gradle` file is not present in the plugin directory the default value will be the name of the directory.

Create the Plugin

Define a class in the `src/main/java/org/sample/MyPlugin.java` implementing the `Plugin` interface.

```
import org.gradle.api.Plugin;
import org.gradle.api.Project;

public class MyPlugin implements Plugin<Project> {

    @Override
    public void apply(Project project) {
         project.getTasks().create("myTask", MyTask.class);
    }

}
```

Define the task extending the `DefaultTask` class:

```
import org.gradle.api.DefaultTask;
import org.gradle.api.tasks.TaskAction;

public class MyTask extends DefaultTask {

    @TaskAction
    public void myTask() {
        System.out.println("Hello World");
    }
}
```

Plugin Class declaration

In the `META-INF/gradle-plugins` folder you have to create a properties file defining the `implementation-class` property that identifies the Plugin implementation class.

In the `META-INF/gradle-plugins/testplugin.properties`

```
implementation-class=org.sample.MyPlugin.java
```

Notice that the **properties filename matches the plugin id**.

How to build and publish it

Change the `build.gradle` file adding some info to upload the plugin in a maven repo:

```
apply plugin: 'java'
apply plugin: 'maven'

dependencies {
    compile gradleApi()
}

repositories {
    jcenter()
}


group = 'org.sample'
version = '1.0'

uploadArchives {
    repositories {
        mavenDeployer {
        repository(url: mavenLocal().url)
        }
    }
}
```

You can build and publish the Gradle plug-in to the Maven repo defined in the `plugin/build.gradle` file using the following command.

```
$ ./gradlew clean uploadArchives
```

How to use it

To use the plugin add in the `build.gradle` of your project:

```
buildscript {
     repositories {
         mavenLocal()
     }
 dependencies {
    classpath group: 'org.sample',    // Defined in the build.gradle of the plugin
              name: 'myplugin',       // Defined by the rootProject.name
              version: '1.0'
    }
 }

apply plugin: 'testplugin'            // Defined by the properties filename
```

Then you can call the task using:

```
 $ ./gradlew myTask
```

Read Gradle Plugins online: https://riptutorial.com/gradle/topic/1900/gradle-plugins

Chapter 7: Gradle Wrapper

Examples

Gradle Wrapper and Git

As discussed in the introduction, the gradle wrapper functionality works because a jar is downloaded into the project to be used when the `gradlew` command is run. However this may not get committed and after the next time the project is checked out, `gradlew` will fail to run with the error:

```
Error: Could not find or load main class org.gradle.wrapper.GradleWrapperMain
```

This will be because your .gitignore will likely include `*jar` for Java projects. When the gradle wrapper was initialised, it copies to the file `gradle/wrapper/gradle-wrapper.jar`. Thus you need to add it to the git index and commit it. Do so with:

```
git add -f gradle/wrapper/gradle-wrapper.jar
git ci
```

With the `-f` being to force it.

Gradle Wrapper introduction

Gradle has the ability to add a wrapper to projects. This wrapper alleviates the need for all users or continuous integration systems to have Gradle installed. It also prevents version issues where there is some incompatibility between the version the project uses and that which users have installed. It does this by installing a version of gradle locally in the project.

Users of the project simply run:

```
> ./gradlew <task> # on *Nix or MacOSX
> gradlew <task>   # on Windows
```

To setup a project to use a wrapper, developers:

1. Execute:

```
gradle wrapper [--gradle-version 2.0]
```

Where `--gradle-version X` is optional and if not provided (or the wrapper task isn't included, as shown below), the version used is the version of gradle being used.

1. To force the project to use a specific version, add the following to the `build.gradle`:

```
task wrapper(type: Wrapper) {
    gradleVersion = '2.0'
}
```

When the `gradle wrapper` command is run it creates the files:

```
the_project/
  gradlew
  gradlew.bat
  gradle/wrapper/
    gradle-wrapper.jar
    gradle-wrapper.properties
```

The official documentation on this feature is at
https://docs.gradle.org/current/userguide/gradle_wrapper.html.

Use locally served Gradle in the Gradle Wrapper

If you want to keep on-premises copy of the Gradle and let the Wrapper use it in the builds, you can set the `distributionUrl` pointing to your copy on the `wrapper` task:

```
task wrapper(type: Wrapper) {
    gradleVersion = '2.0'
    distributionUrl = "http\://server/dadada/gradle-${gradleVersion}-bin.zip"
}
```

after executing `gradle wrapper`, the shell script `gradlew` is created and the `gradle/wrapper/gradle-wrapper.properties` is configured to use provided URL to download the Gradle.

Using the Gradle Wrapper behind a proxy

The first time a user runs a project's `gradlew`, it should be realized that it will do two key things:

1. Check to see if the version of the gradle used by the wrapper is already in ~/.gradle/wrapper/dists
2. If not, download the archive of the version from the internet

If you're in an environment that requires all external traffic to go through a proxy, step two is going to fail (unless it's a transparent proxy environment). As a result, you need to ensure your have the *JVM* proxy parameters set.

For example, if you have a basic proxy setup with no authentication, simply set the environment variable `JAVA_OPTS` or `GRADLE_OPTS` with:

```
-Dhttps.proxyPort=<proxy_port> -Dhttps.proxyHost=<hostname>
```

So a completed example on windows would be:

```
set JAVA_OPTS=-Dhttps.proxyPort=8080 -Dhttps.proxyHost=myproxy.mycompany.com
```

If however your environment also requires authentication, then you'll also want to review your other options at https://docs.oracle.com/javase/8/docs/api/java/net/doc-files/net-properties.html.

*NOTE: This proxy configuration is in **addition** to any proxy configuration for your dependency repository access.*

Read Gradle Wrapper online: https://riptutorial.com/gradle/topic/3006/gradle-wrapper

Chapter 8: Including Native Source - Experimental

Parameters

Parameters	Details
model.android.ndk.toolchain	native toolchain found in the ndk-bundle folder

Examples

Basic JNI Gradle Config

root: build.gradle

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle-experimental:0.8.0-alpha4'
    }
}

allprojects {
    repositories {
        jcenter()
    }
}
```

app: build.gradle

```
apply plugin: 'com.android.model.application'

dependencies {
    compile "com.android.support:support-v4:23.3.0"
    compile fileTree(dir: 'libs', include: '*.jar')
}

model {
    android {
        compileSdkVersion = 23
        buildToolsVersion = '23.0.3'

        defaultConfig {
            applicationId = 'com.example.hello'
            minSdkVersion.apiLevel = 9
            targetSdkVersion.apiLevel = 23

            buildConfigFields {
                create() {
                    type "int"
                    name "VALUE"
                    value "1"
                }
            }
        }

        ndk {
            platformVersion = 9
            moduleName "hello"

            toolchain "clang"

            stl "gnustl_static"
            CFlags.add("-DANDROID_NDK")
            cppFlags.add("-std=c++11")

            ldLibs.add("android")
            ldLibs.add("dl")
            ldLibs.add("log")
        }

        sources {
            main {
                jni {
                    exportedHeaders {
                        srcDirs "../../common/headers"
                    }
                    source {
                        srcDirs "../../common/src"
                    }
                }
            }
        }
    }
}
```

Using prebuilt libraries and OpenGL ES 2.0

root: build.gradle

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle-experimental:0.8.0-alpha4'
    }
}

allprojects {
    repositories {
        jcenter()
    }
}
```

app: build.gradle

```
apply plugin: 'com.android.model.application'

dependencies {
    compile "com.android.support:support-v4:23.3.0"
    compile fileTree(dir: 'libs', include: '*.jar')
}

model {
    android {
        compileSdkVersion = 23
        buildToolsVersion = '23.0.3'

        defaultConfig {
            applicationId = 'com.example.glworld'
            minSdkVersion.apiLevel = 9
            targetSdkVersion.apiLevel = 23

            buildConfigFields {
                create() {
                    type "int"
                    name "VALUE"
                    value "1"
                }
            }
        }

        buildTypes {
            release {
                minifyEnabled = false
                proguardFiles.add(file('proguard-rules.txt'))
            }
        }

        ndk {
            platformVersion = 9
            moduleName "glworld"

            toolchain "clang"

            stl "gnustl_static"
            CFlags.add("-DANDROID_NDK")
            CFlags.add("-DDISABLE_IMPORTGL")
            CFlags.add("-DFT2_BUILD_LIBRARY=1")
            cppFlags.add("-std=c++11")

            ldLibs.add("EGL")
            ldLibs.add("android")
            ldLibs.add("GLESv2")
            ldLibs.add("dl")
            ldLibs.add("log")
        }

        sources {
            main {
                jni {
                    dependencies {
                        library "freetype2" linkage "shared"
                    }
                    exportedHeaders {
                        srcDirs "../../common/headers"
                    }
                    source {
                        srcDirs "../../common/src"
                    }
                }
            }
        }
    }

    repositories {
        prebuilt(PrebuiltLibraries) {
            freetype2 {
                headers.srcDir "../../common/freetype2-android/include"
                binaries.withType(SharedLibraryBinary) {
                    def localLib = "../../common/freetype2-android/Android/libs"
                    sharedLibraryFile =
                            file("$localLib/${targetPlatform.getName()}/libfreetype2.so")
                }
            }
        }
    }
}

// The next tasks compile a freetype library using a make file.
// These `.so`'s are then used as the shared libraries compiled above.
tasks.withType(JavaCompile) {
    compileTask -> compileTask.dependsOn buildNative
}

// Call regular ndk-build (.cmd) script from the app directory
task buildNative(type: Exec) {
    def ndkDir = "/Development/android-sdk-macosx/ndk-bundle"
    commandLine "$ndkDir/ndk-build",
            '-C',
            file('../../common/freetype2-android/Android/jni').absolutePath
}

task cleanNative(type: Exec) {
    def ndkDir = "/Development/android-sdk-macosx/ndk-bundle"
    commandLine "$ndkDir/ndk-build",
            '-C',
            file('../../common/freetype2-android/Android/jni').absolutePath,
            "clean"
}

clean.dependsOn cleanNative
```

Read Including Native Source - Experimental online:
https://riptutorial.com/gradle/topic/4460/including-native-source---experimental

Chapter 9: Initializing Gradle

Remarks

Terminology

- Task - an atomic piece of work which a build performs. Tasks have `inputs`, `outputs` and task dependencies.
- `dependencies {}` - Declares `File` or binary dependencies necessary to execute tasks. For example, `org.slf4j:slf4j-api:1.7.21` is shorthand coordinates to a Maven dependency.
- `repositories {}` - How Gradle finds files for external dependencies. Really, just a collection of files organized by group, name, and version. For example: `jcenter()` is a convenience method for `maven { url 'http://jcenter.bintray.com/' } }`, a Bintray Maven repository.

Examples

Initializing a New Java Library

Prerequisite: Installing Gradle

Once you have Gradle installed, you can setup a new or existing project by running

```
cd $PROJECT_DIR
gradle init --type=java-library
```

Note that there are other project types like Scala you can get started with, but we'll use Java for this example.

You will end up with:

```
.
├── build.gradle
├── gradle
│   └── wrapper
│       ├── gradle-wrapper.jar
│       └── gradle-wrapper.properties
├── gradlew
├── gradlew.bat
├── settings.gradle
└── src
    ├── main
    │   └── java
    │       └── Library.java
    └── test
        └── java
            └── LibraryTest.java
```

You can now run `gradle tasks` and see that you can build a `jar`, run `test`s, produce `javadoc`s and much more even though your `build.gradle` file is:

```
apply plugin: 'java'

repositories {
    jcenter()
}

dependencies {
    compile 'org.slf4j:slf4j-api:1.7.21'
    testCompile 'junit:junit:4.12'
}
```

Read Initializing Gradle online: https://riptutorial.com/gradle/topic/2247/initializing-gradle

Chapter 10: IntelliJ IDEA Task Customization

Syntax

- groovy.util.Node = node.find { childNode -> return true || false }
- node.append(nodeYouWantAsAChild)
- groovy.util.Node parsedNode = (new XmlParser()).parseText(someRawXMLString)
- ''' mutli-line string (not interpolated) '''

Remarks

The three basic files of an IntelliJ project - the ipr, iws, and iml files - can be accessed as in gradle in the idea task through

```
project.ipr
module.iml
workspace.iws
```

using the .withXml lets you access the xml. Using the .asNode() on that turns it into a groovy xml node.

Ex:

```
project.ipr.withXml { provider ->
    def node = provider.asNode()
```

From there it's pretty simple - to modify gradle to configure IntelliJ projects for you, take the file as it starts, perform the actions you'd like gradle to take (inside IntelliJ), and then diff the new file with the old file. You should see what XML you'll need to customize the idea job. You'll also need to take note of where in the xml it's located.

One other thing to consider is that you don't want duplicate nodes within the IntelliJ files if you run the gradle idea multiple times. So, you'll want to search for the node you'd like to make and if it's not there, you can create and insert it.

Pitfalls:

Sometimes, when using == for string comparison in the find method, it fails. When testing and I find that to be the case, I use .contains.

When searching for nodes, not all nodes have the attribute you're using as a criteria, so be sure to check for null.

Examples

Add a Basic Run Configuration

Assumptions for this example:

- You have a class, `foo.bar.Baz`.
- You'd like to create a run configuration that runs the main method.
- It's in a module called `fooBar`.

In your gradle file:

```
idea {
    workspace.iws.withXml { provider ->
        // I'm not actually sure why this is necessary
        def node = provider.asNode()

        def runManager = node.find { it.@name.contains('RunManager')}

        // find a run configuration if it' there already
        def runner = runManager.find { it.find ({ mainClass ->
            return mainClass.@name != null && mainClass.@name == "MAIN_CLASS_NAME" &&
mainClass.@value != null && mainClass.@value.contains('Baz');
        }) != null }

        // create and append the run configuration if it doesn't already exists
        if (runManager != null && runner == null){
            def runnerText = '''
                <configuration default="false" name="Baz" type="Application"
factoryName="Application" nameIsGenerated="true">
                  <extension name="coverage" enabled="false" merge="false" runner="idea">
                    <pattern>
                      <option name="PATTERN" value="foo.bar.Baz" />
                      <option name="ENABLED" value="true" />
                    </pattern>
                  </extension>
                  <option name="MAIN_CLASS_NAME" value="foo.bar.Baz" />
                  <option name="VM_PARAMETERS" value="" />
                  <option name="PROGRAM_PARAMETERS" value="" />
                  <option name="WORKING_DIRECTORY" value="file://$PROJECT_DIR$" />
                  <option name="ALTERNATIVE_JRE_PATH_ENABLED" value="false" />
                  <option name="ALTERNATIVE_JRE_PATH" />
                  <option name="ENABLE_SWING_INSPECTOR" value="false" />
                  <option name="ENV_VARIABLES" />
                  <option name="PASS_PARENT_ENVS" value="true" />
                  <module name="foobar" />
                  <envs />
                  <method />
                </configuration>'''
            runner = (new XmlParser()).parseText(runnerText)
            runManager.append(config);
        }

        // If there is no active run configuration, set the newly made one to be it
        if (runManager != null && runManager.@selected == null) {
            runManager.@selected="${runner.@factoryName}.${runner.@name}"
        }
    }
}
```

Read IntelliJ IDEA Task Customization online: https://riptutorial.com/gradle/topic/2297/intellij-idea-task-customization

Chapter 11: Ordering tasks

Remarks

> Please note that `mustRunAfter` and `shouldRunAfter` are marked as "incubating" (as of Gradle 3.0) which means that these are experimental features and their behavior can be changed in future releases.

There are two ordering rules available:

- `mustRunAfter`
- `shouldRunAfter`

When you use the `mustRunAfter` ordering rule you specify that taskB must always run after taskA, whenever both taskA and taskB will be run.

The `shouldRunAfter` ordering rule is similar but less strict as it will be ignored in two situations:

- if using that rule introduces an ordering cycle.
- when using parallel execution and all dependencies of a task have been satisfied apart from the `shouldRunAfter` task, then this task will be run regardless of whether its `shouldRunAfter` dependencies have been run or not.

Examples

Ordering with the mustRunAfter method

```
task A << {
    println 'Hello from A'
}
task B << {
    println 'Hello from B'
}

B.mustRunAfter A
```

The `B.mustRunAfter A` line tells Gradle to run task after task specified as an argument.

And the output is:

```
> gradle -q B A
Hello from A
Hello from B
```

The ordering rule doesn't introduce dependency between the A and the B tasks, but has an effect only when **both tasks are scheduled** for execution.

It means that we can execute tasks A and B independently.

The output is:

```
> gradle -q B
Hello from B
```

Read Ordering tasks online: https://riptutorial.com/gradle/topic/5550/ordering-tasks

Chapter 12: Task dependencies

Remarks

doLast

Note, that in a gradle 3.x more idiomatic way task definition: using **explicit doLast{closure}** notation instead "leftShift"(<<) operator preferable.(**leftShift** has been deprecated in a gradle 3.2 is scheduled to be removed in gradle 5.0.)

```
task oldStyle << {
    println 'Deprecated style task'
 }
```

is equivalent to:

```
task newStyle {
    doLast {
    println 'Deprecated style task'
    }
 }
```

Examples

Adding dependencies using task names

We can change the tasks execution order with the `dependsOn` method.

```
task A << {
    println 'Hello from A'
}
task B(dependsOn: A) << {
    println "Hello from B"
}
```

Adding `dependsOn: causes:

- task B depends on task A
- Gradle to execute `A` task everytime **before** the `B` task execution.

And the output is:

```
> gradle -q B
Hello from A
Hello from B
```

Adding dependencies from another project

```
project('projectA') {
    task A(dependsOn: ':projectB:B') << {
        println 'Hello from A'
    }
}

project('projectB') {
    task B << {
        println 'Hello from B'
    }
}
```

To refer to a task in another project, you **prefix the name of the task** with the path of the project it belongs to `:projectB:B`.

And the output is:

```
> gradle -q B
Hello from A
Hello from B
```

Adding dependency using task object

```
task A << {
    println 'Hello from A'
}

task B << {
    println 'Hello from B'
}

B.dependsOn A
```

It is an alternative way to define the dependency instead of using the task name.

And the output is the same:

```
> gradle -q B
Hello from A
Hello from B
```

Adding multiple dependencies

You can add multiple dependencies.

```
task A << {
    println 'Hello from A'
}

task B << {
    println 'Hello from B'
}
```

```
task C << {
    println 'Hello from C'
}

task D << {
    println 'Hello from D'
}
```

Now you can define a set of dependencies:

```
B.dependsOn A
C.dependsOn B
D.dependsOn C
```

The output is:

```
> gradle -q D
Hello from A
Hello from B
Hello from C
Hello from D
```

Other example:

```
B.dependsOn A
D.dependsOn B
D.dependsOn C
```

The output is:

```
> gradle -q D
Hello from A
Hello from B
Hello from C
Hello from D
```

Multiple dependencies with the dependsOn method

You can add multiple dependencies.

```
task A << {
    println 'Hello from A'
}

task B(dependsOn: A) << {
    println 'Hello from B'
}

task C << {
    println 'Hello from C'
}

task D(dependsOn: ['B', 'C'] << {
    println 'Hello from D'
}
```

The output is:

```
> gradle -q D
Hello from A
Hello from B
Hello from C
Hello from D
```

Read Task dependencies online: https://riptutorial.com/gradle/topic/5545/task-dependencies

Chapter 13: Using third party plugins

Examples

Adding a third party plugin to build.gradle

Gradle (All Versions) *This method works for all versions of gradle*

Add the buildscript code at the beginning of your build.gradle file.

```
buildscript {
  repositories {
    maven {
      url "https://plugins.gradle.org/m2/"
    }
  }
  dependencies {
    classpath "org.example.plugin:plugin:1.1.0"
  }
}

apply plugin: "org.example.plugin"
```

Gradle (Versions 2.1+) *This method only works for projects using Gradle 2.1 or later.*

```
plugins {
  id "org.example.plugin" version "1.1.0"
}
```

build.gradle with multiple third party plugins

Gradle (All Versions)

When adding multiple third party plugins you do not need to separate them into different instances of the buildscript(All) or plugin(2.1+) code, new plug ins can be added alongside pre-existing plugins.

```
buildscript {
  repositories {
    maven {
      url "https://plugins.gradle.org/m2/"
    }
  }
  dependencies {
    classpath "org.example.plugin:plugin:1.1.0"
    Classpath "com.example.plugin2:plugin2:1.5.2"
  }
}

apply plugin: "org.example.plugin"
apply plugin: "com.example.plugin2"
```

Gradle (Versions 2.1+)

```
plugins {
  id "org.example.plugin" version "1.1.0"
  id "com.example.plugin2" version "1.5.2"
}
```

Read Using third party plugins online: https://riptutorial.com/gradle/topic/9183/using-third-party-plugins

Credits

S. No	Chapters	Contributors
1	Getting started with gradle	Afterfield, bassim, Community, Emil Burzo, Eric Wendelin, Hamzawey, Hillkorn, Matthias Braun, Nikem, Pepper Lebeck-Jobe, Sergey Yakovlev, Stanislav, user2555595, vanogrid, Will
2	Auto Increment Version Number Using Gradle Script For Android Applications	Jayakrishnan PM
3	Dependencies	Afshin, Andrii Abramov, GameScripting, Hillkorn, leeor, Matthias Braun, mcarlin, mszymborski, Will
4	Gradle Init Scripts	ambes, Hillkorn
5	Gradle Performance	ambes, Sergey Yakovlev, Will
6	Gradle Plugins	Gabriele Mariotti, JBirdVegas
7	Gradle Wrapper	ajoberstar, Fanick, HankCa, I Stevenson
8	Including Native Source - Experimental	iHowell
9	Initializing Gradle	Eric Wendelin, Will
10	IntelliJ IDEA Task Customization	IronHorse, Sam Sieber, Will
11	Ordering tasks	Gabriele Mariotti
12	Task dependencies	Gabriele Mariotti, Sergey Yakovlev, Stanislav
13	Using third party plugins	Afterfield