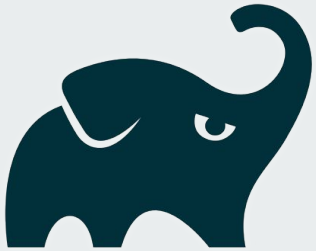


Gradle

CS 527 @ Fall 2017, UIUC

Han Yin, Hill Randolph William, Lloyd Ramey

Presentation Contents

- **WHAT**:
  - Gradle as a build automation system
- **WHY**:
  - Comparison between Gradle and other competitors
- **HOW**:
  - Project structure, Gradle scripts, Plugins
  - Intro to course projects

[WHAT]
Gradle: Build Automation System

Softwares builds are cumbersome

- Dependency resolution for compiling source & test codes
  - Different forms: files, external libraries, artifacts, urls, etc.
  - Different versions for each dependency
- Continuous integration (CI)
  - Running test suites with different configurations
  - Generating reports
- Continuous delivery (CD)
  - Packaging compiled files into compressed formats, e.g. *Jar*, *Apk*, etc.
  - Deployment to servers, publishing releases

Case Study

- The Performance of a build system has a huge impact on the **cost and engineering efficiency**:

  “In 2017, we worked with a 600-engineer team.
  On average, the engineers each ran about 42 builds per week.
  The team estimated the cost per-minute per engineer to be US$1.42,
  and they work at least 44 weeks annually.”

  https://gradle.org/gradle-vs-maven-performance/

- **Build Automation Systems** are invented to ease the pain of intensive repetitive ***tasks***.

Gradle

- JVM based build automation tools:
  - ***Apache Ant***: 2000
  - ***Apache Maven***: 2004
  - ***Gradle***: 2007
- Initially targeted JVM languages:
  - Java, Groovy, Scala
- Plugin based extensibility
  - Android: “com.android.application”
  - C/C++: “c”, “cpp”
- **Declarative builds** and **build-by-convention**
- ***DSL*** for dependency based programming
- Created with **multi-project builds** in mind
- “The first build integration tool”
  - Deep import for ***Ant*** projects / tasks
  - Fully support for existing ***Maven*** or ***Ivy*** repository infrastructure

[WHY]
Gradle vs Ant & Maven

Apache Ant (with Apache Ivy)

The first ‘modern’ build tool targeting the JVM

- Released in 2000
- Configured using ***XML***
- **Highly customizable builds** based on procedural programming
- Initially only allowed local dependencies
- Adopted ***Ivy*** for dependency management over the network

Ant drawbacks

- Developers must write all commands to execute some task
- ***XML*** is not a good fit for procedural programming
- Requires verbose build scripts for simple tasks
- Build scripts tend to become unmanageable for non-trivial projects
- Initially required storing all dependencies alongside code

Ant (with Ivy)

- Highly customizable
- No convention, **only configuration**
- ***XML*** build scripts for procedural programming
- Dependency Management with *Ivy*

Gradle

- Highly customizable
- **Convention over configuration**
- Declarative ***Groovy DSL***, allowing imperative programming if required
- Dependency Management based on *Ivy*

Apache Maven

Aiming at the pain points developers faced when using ***Apache Ant***

- Released in 2004
- Configured using ***XML***
- **Convention over configuration**
- Relies on **plugins** for extending behavior
- Introduced automated dependency downloads
- Standardized build life-cycle

Maven drawbacks

- **Poor dependency version conflict resolution**
- ***XML*** configuration is strictly structured and standardized
- Complex builds can be very complicated to write
- Large project build scripts can have hundreds of lines without doing anything advanced/extraordinary
- Difficult to customize build logic

Maven

- Difficult to customize
- Convention over configuration
- Declarative ***XML*** build scripts
- Dependency management

Gradle

- Highly customizable
- Convention over configuration
- Declarative ***Groovy DSL***, allowing imperative programming if required
- Dependency management based on ***Ivy***

Buildscript showdown

Compiling a simple Java project in the three main JVM build systems

Ant (with Ivy)

ivy.xml:

```
<ivy-module version="2.0">
    <info organisation="com.example" module="building-java-code"/>
    <dependencies>
        <dependency org="junit" name="junit" rev="4.12"/>
        <dependency org="org.hamcrest" name="hamcrest-all" rev="1.3"/>
    </dependencies>
</ivy-module>
```

Ant (with Ivy)

build.xml:

```
<project xmlns:ivy="antlib:org.apache.ivy.ant"
name="building-java-code" default="jar">
    <path id="lib.path.id">
        <fileset dir="lib" />
    </path>
    <target name="resolve">
        <ivy:retrieve />
    </target>
    <target name="clean">
        <delete dir="build"/>
    </target>
```

```
 <target name="compile" depends="resolve">
       <mkdir dir="build/classes"/>
        <javac srcdir=src"
destdir="build/classes"
classpathref="lib.path.id"/>
    </target>
    <target name="jar" depends="compile">
        <mkdir dir="build/jar"/>
        <jar
destfile="build/jar/building-java-code.jar"
basedir="build/classes"/>
    </target>
</project>
```

Maven

pom.xml:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>building-java-code</artifactId>
  <packaging>jar</packaging>
  <version>1.0</version>
  <dependencies>
      <dependency>
          <groupId>junit</groupId>
          <artifactId>junit</artifactId>
          <version>4.12</version>
      </dependency>
    <dependency>
      <groupId>org.hamcrest</groupId>
      <artifactId>hamcrest-all</artifactId>
      <version>1.3</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-compiler-plugin</artifactId>
      <version>2.3.2</version>
    </plugin>
  </plugins>
  </build>
</project>
```

Gradle

build.gradle:

```
apply plugin: 'java'

group = 'com.example'

repositories {
    jcenter()
}

dependencies {
    testCompile 'junit:junit:4.12'
    testCompile 'org.hamcrest:hamcrest-all:1.3'
}
```

[HOW]
Write Gradle scripts

Gradle project structure

Sample of a standard root project

```
.
+-- build/
+-- buildSrc/
+-- build.gradle
+-- settings.gradle
|
+-- sub-project-1/
|   +-- …
+-- sub-project-2/
|   +-- …
|
+-- gradle/
|   +-- wrapper/
|       +-- gradle-wrapper.jar
|       +-- gradle-wrapper.properties
+-- gradlew
+-- gradlew.bat
```

Multi-project Builds

- Specify sub-projects in **`settings.gradle`**:

```
include 'sub-project-1', 'sub-project-2'
```

- By default, the build file is **`build.gradle`**.
  - Each sub-project has its own build script.

```
...
+-- settings.gradle
|
+-- build/
+-- buildSrc/
+-- build.gradle
|
+-- sub-project-1/
|   +-- build/
|   +-- build.gradle
|
+-- sub-project-2/
|   +-- build/
|   +-- build.gradle
...
```

Gradle Wrapper

- Ensures that the **correct gradle version** is always used for the project.
- Generated by running the built-in ‘wrapper’ task.
  `> gradle wrapper`
- Wrapper files should be **committed** into VCS.
- Downloads the correct version the first time it’s run.

```
.
+-- build/
+-- buildSrc/
+-- build.gradle
+-- settings.gradle
|
+-- sub-project-1/
|   +-- …
+-- sub-project-2/
|   +-- …
|
+-- gradle/
|   +-- wrapper/
|       +-- gradle-wrapper.jar
|       +-- gradle-wrapper.properties
+-- gradlew
+-- gradlew.bat
```

Gradle Scripts

Sample of a Gradle build script

build.gradle:

```
apply plugin: 'java'

def pkgName = 'com.example'

group = pkgName

repositories {
    jcenter()
}

dependencies {
    testCompile 'junit:junit:4.12'
    testCompile
'org.hamcrest:hamcrest-all:1.3'
}
```

Gradle scripts

- First, Gradle scripts are configuration scripts.
  - As a script executes, it configures a **Delegate Object** of itself.
  - Then, the properties and methods of the **Delegate Object** are available to use in the script.
- Delegate types:
  - Build script: `build.gradle` **Project**
  - Init script: `init.gradle` **Gradle**
  - Settings script: `settings.gradle` **Settings**
- Second, Each *Gradle* script implements the ***Script*** interface, which defines a number of properties and methods which you can use in the script.

Gradle scripts: Delegate objects

build.gradle:

```
apply plugin: 'java'

group = 'com.example'

repositories {
    jcenter()
}

dependencies {
    testCompile 'junit:junit:4.12'
    testCompile 'org.hamcrest:hamcrest-all:1.3'
}

test {
      reports.html.enabled = false
}
```

- Delegate Object: *Project*
  - group
  - apply()
  - repositories()
  - dependencies()
- Plugin: *Java*
  - test()
  - Dependency configurations:
    - junit, hamcrest...

```
> Could not find method compile() for arguments [junit:junit:4.12]
 on object of type org.gradle.api.internal.artifacts.dsl.dependenc
ies.DefaultDependencyHandler.
```

Gradle scripts: Build script structure

- A build script is made up of zero or more **Statements** and **Script Blocks**.
  - **Statements** can include:
    - local variable declarations, method calls, property assignments.
  - A **Script Block** is a **method call** which takes a closure as a parameter. The closure is treated as a configuration closure which configures the **Delegate Object** as it executes.

Gradle scripts: Statements

build.gradle:

```
apply plugin: 'java'

def pkgName = 'com.example'

group = pkgName

repositories {
    jcenter()
}

dependencies {
    testCompile 'junit:junit:4.12'
    testCompile 'org.hamcrest:hamcrest-all:1.3'
}
```

local variable declaration

property assignment

method call

script block ? ? ?

Gradle DSL

- *Gradle* script is also a *Groovy* script, therefore it can contain those elements allowed in *Groovy*:
  - **Method definitions**, **Class definitions**, **Annotations**, **Comments**, etc.

```
/* Get password from Mac OS Keychain */
def getPassword(String keyChain) {
   def stdout = new ByteArrayOutputStream()
   def stderr = new ByteArrayOutputStream()
   exec {
       commandLine 'security', 'find-generic-password',
keyChain
       standardOutput = stdout
       errorOutput = stderr
       ignoreExitValue true
   }
   stderr.toString().trim().drop(10).replaceAll('"','')
}
```

```
// A customized task to print a greeting
class GreetingTask extends DefaultTask {
    @TaskAction
    def greet() {
        println 'Greetings!'
    }
}

// Create a task using the task type
task hello(type: GreetingTask)
```

Gradle scripts: Script blocks

- A **script block** is a **method call** which takes a closure as a parameter.

```
dependencies()
```

```
dependencies({
    testCompile 'junit:junit:4.12'
    testCompile 'org.hamcrest:hamcrest-all:1.3'
})
```

A method call that takes a closure as a parameter.

```
dependencies(){
    ......
}
```

Since the closure is the last parameter,
it can be moved out of the parentheses.

```
dependencies {
    ......
}
```

Parentheses can be omitted **if no ambiguity**:
A script block!

Gradle Tasks

Customize the Gradle build process with your own tasks

- Everything in *Gradle* sits on top of two basic concepts:
  - **Projects**
  - **Tasks**

Gradle Tasks

- A **Project** includes a collection of **Tasks**.
- A ***Task*** represents a single atomic piece of work for a build.
  - E.g. compiling classes, generating javadoc, etc.
- Each task belongs to a ***Project***, and has a name for reference within this project.
- A task may have dependencies on other tasks, or might be scheduled to always run after another task.

Gradle Tasks

- To view the tasks in ***groups***:
  - `gradle tasks`
  - `gradle tasks --all`
- Gradle supports two types of task:
  - One is the **simple task**
  - The other is the **enhanced task**

```
ins-iMac:test naco_siren$ gradle tasks --all

> Task :tasks

------------------------------------------------------------
All tasks runnable from root project
------------------------------------------------------------

Build Setup tasks
-----------------
init - Initializes a new Gradle build.
wrapper - Generates Gradle wrapper files.

Help tasks
----------
buildEnvironment - Displays all buildscript dependencies declared in root project 'test'.
components - Displays the components produced by root project 'test'. [incubating]
dependencies - Displays all dependencies declared in root project 'test'.
dependencyInsight - Displays the insight into a specific dependency in root project 'test'
.
dependentComponents - Displays the dependent components of components in root project 'tes
t'. [incubating]
help - Displays a help message.
model - Displays the configuration model of root project 'test'. [incubating]
projects - Displays the sub-projects of root project 'test'.
properties - Displays the properties of root project 'test'.
tasks - Displays the tasks runnable from root project 'test'.

Other tasks
-----------
build_528oto0kyvtt7ukrwpumfhqg2$_run_closure8@5bb1341a
myTask1
myTask2
myTask3
myTask4
myTask5
myTask6
myTask7
myTask8


BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

Gradle Tasks: Simple Task

- One is the **simple task**:
  - You define the task with an ***action closure*** to determine the behaviour of the task.
  - Good for implementing one-off tasks in your build script.

```
task taskX(dependsOn: 'taskY'){
    doLast {
        println 'taskX'
    }
}

task taskY {
    doLast {
        println 'taskY'
    }
}

> gradle taskX
```

Gradle Tasks: Simple Task

- Code specified in ***doLast*** will be run after all other task actions have been executed
- Code specified in ***doFirst*** will be run before all other task actions have been executed
- Tasks can be optionally skipped if ***onlyIf*** closure evaluates to false

```
task hello {
    doLast {
        println 'Hello World'
    }
    doFirst {
        if(!usingEclipse) {
            throw new StopExecutionException()
        }
    }
    onlyIf {
        project.hasProperty('usingEclipse')
    }
}
```

Gradle Tasks: Enhanced Task

- The other is the **enhanced task**:
  - The behaviour is built into the task, and the task provides some properties.
  - You don't need to implement the task behaviour like simple tasks, but simply declare the task and configure the task using its properties.
  - Most Gradle plugins use enhanced tasks.

```
task copy(type: Copy) {
    from 'resources'
    into 'target'
    include('**/*.txt', '**/*.xml',
'**/*.properties')
}
```

Gradle Build Lifecycle

- There are three phases in a build:
  - **Initialization**
  - **Configuration**
  - **Execution**

Gradle Build Lifecycle: Initialization

- Gradle determines which projects are going to take part in the build, and creates a ***Project*** instance for each of these projects.
- ***Settings.gradle*** is evaluated if present.
  - *A multi-project build must have this file in the root project!*
  - This is the only place to set the root project name, otherwise it defaults to the folder name.
  - Can reference gradle properties.
  - Can modify plugin resolution for projects.

Gradle Build Lifecycle: Configuration

- Gradle configure the project objects on "*configuration on demand*" mode.
  - Only executes the ***build.gradle*** file of projects that are participating in the build.
  - Provides a performance boost for large multi-project builds.
- All the tasks are configured during this stage.
  - But the ***action scripts*** are NOT NECESSARILY executed.
    - i.e. *doFirst*, *doLast*, *onlyIf*, etc.

Gradle Build Lifecycle: Execution

- Gradle determines the subset of tasks to execute.
  - Determined by the task names passed to the gradle command
- Gradle then executes each of the selected tasks.
  - Execution is ordered by dependencies
  - Parallel task execution is an *incubating* feature
  - Any task failures will cause the build to fail and stop prematurely

```
group = artifact_group
version = "${artifact_version}.${build_number}"

apply plugin: 'groovy'
apply plugin: 'maven'

repositories {
        mavenCentral()
}

dependencies {
    compile gradleApi()
    compile localGroovy()
    testCompile gradleTestKit()
}

task wrapper(type: Wrapper) {
    gradleVersion = "2.10"
}

test {
    // Always run the tests
    outputs.upToDateWhen { false }

    // Turn on some console logging
    testLogging {
        events "passed", "skipped", "failed"
        exceptionFormat "full"
        stackTraceFilters "entryPoint"
    }
}
```

```

// Write the plugin's classpath to a file to share with the tests
task createClasspathManifest {
    def outputDir = file("$buildDir/$name")

    inputs.files sourceSets.main.runtimeClasspath
    outputs.dir outputDir

    doLast {
        outputDir.mkdirs()
        file("$outputDir/plugin-classpath.txt").text = sourceSets.main.runtimeClasspath.join("\n")
    }
}

// Add the classpath file to the test runtime classpath
dependencies {
    testRuntime files(createClasspathManifest)
}

sourceCompatibility = 1.7
targetCompatibility = 1.7

```

A sample Java build script

https://github.com/uber-common/infer-plugin/blob/master/infer-plugin/build.gradle

Gradle Plugins

- **Extend *Gradle DSL*** by adding capabilities to *Gradle* projects
- Large collection of plugins built into *Gradle*
- Many third-party plugins available

Java Plugin

- `apply plugin: 'java'`
- The java plugin allows Gradle to compile java source, as well as run *JUnit* / *TestNG* based tests and bundle build artifacts.
- It also serves as the basis for many other Gradle plugins.

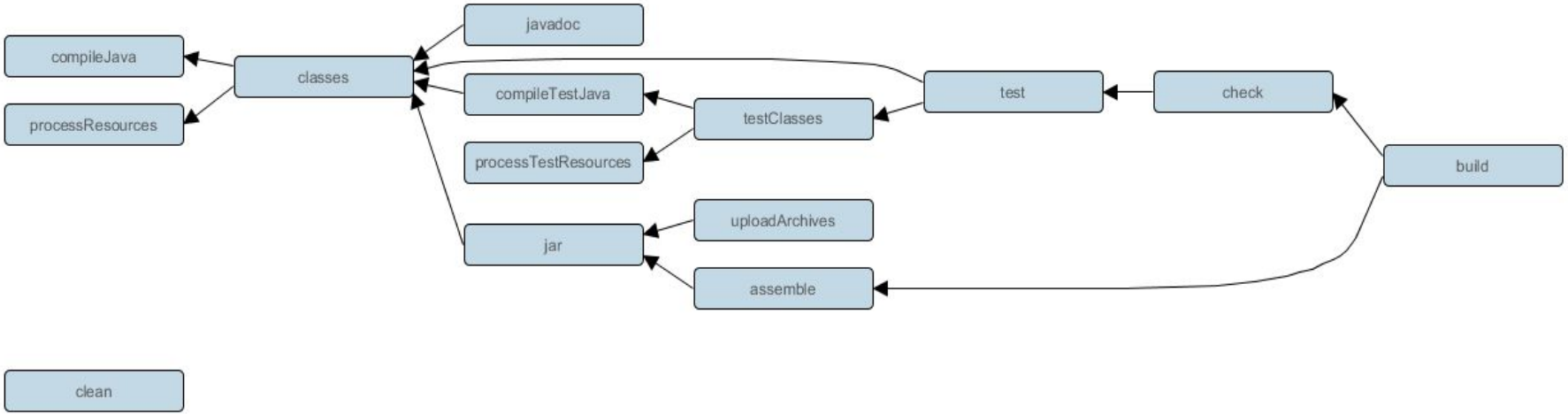
Java Plugin: Project Structure

- *Gradle* uses the same directory conventions as *Maven*, i.e. convention over configuration
- Configurable in the build script's ***sourceSets***:

```
sourceSets {
    main {
        java {
            srcDirs = ['src/java']
        }
        resources {
            srcDirs = ['src/resources']
        }
    }
}
```

Directory	Meaning
`src/main/java`	Production Java source
`src/main/resources`	Production resources
`src/test/java`	Test Java source
`src/test/resources`	Test resources
`src/`*`sourceSet`*`/java`	Java source for the given source set
`src/`*`sourceSet`*`/resources`	Resources for the given source set

Java Plugin: Task Depencencies



Java Plugin: Dependency Management

- You must specify which ***repositories*** to pull dependencies from.
- Works with either *Maven* repositories or *Ivy* repositories.
- Provides shortcuts for the two major *Maven* repositories.
  - MavenCentral
  - JCenter
- Easy to specify custom repositories.

```
apply plugin: 'java'

...
repositories {
    mavenCentral()
    jcenter()
    maven {
        url 'http://my-repo/'
    }
    ivy {
        url 'http://my-repo/'
    }
}
```

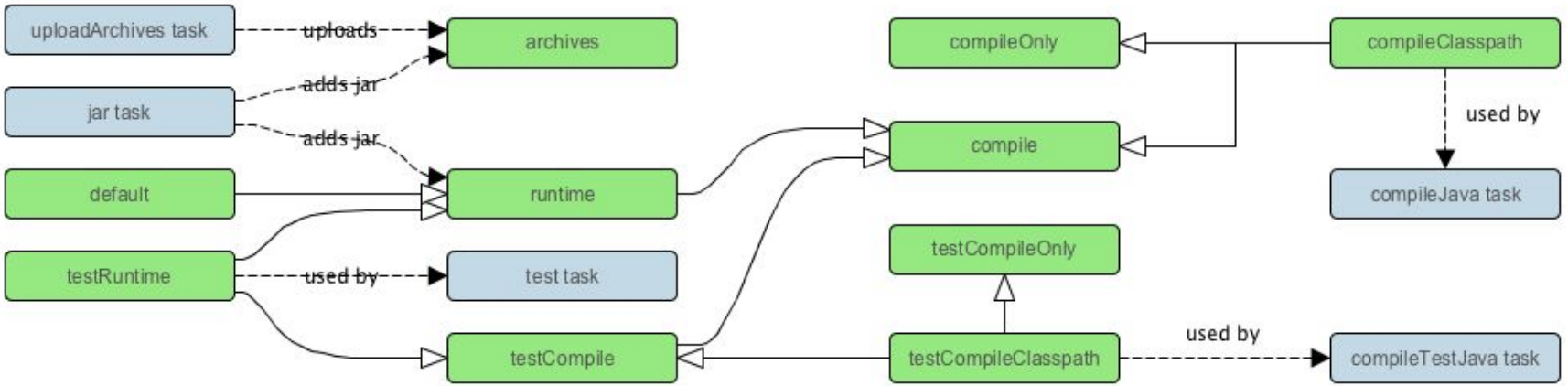
Java Plugin: Dependency Management

- *Maven* dependencies are specified using:
  - Group ID, Artifact ID, Version
- Local *jars* can be included using file collections
- Projects included by passing a reference to the subproject

```
apply plugin: 'java'

...
dependencies {
    compile 'commons-io:commons-io:2.6'
    testCompile 'junit:junit:4.12'
    runtime fileTree(dir:'libs', include
'*.jar')
    testRuntime files('libs/a.jar')
    compile project(':other')
}
```

Java Plugin: Dependency Management



Android Plugin

- `apply plugin: 'com.android.application'`
- The Android plugin adds functionality to Gradle for building, testing, and releasing android applications independently of *Android Studio / Eclipse*.
- It is a third-party plugin developed by *Google*, and typically updated in lock-step with *Android Studio*.
- As the *Android* plugin isn't built into gradle, and Google does not publish it to Gradle's plugin portal, you have to tell gradle where to find the plugin.
- Gradle 4.1 and higher includes the 'google()' repository shortcut.

Other Common Plugins

- `java-base`
  - Adds *Java* compilation, testing and bundling capabilities to a project.
  - It serves as the basis for many of the other Gradle plugins.
- `java, groovy-base`
  - Adds support for building *Groovy* projects.
- `java, scala-base`
  - Adds support for building *Scala* projects.

Custom Plugin

- `apply plugin: 'my.custom.plugin'`
- Gradle allows you to implement your own plugins, so you can reuse your build logic, and share it with others
- You can implement a *Gradle* plugin in any language that ends up compiled as bytecode.
- ***Gradle API*** has been designed to work well with *Groovy*, *Java* or *Kotlin*

Custom Plugin: Packaging

- Build Script
  - Can include the source for the plugin directly in the build script.
  - Pro: plugin is automatically compiled and included in the classpath of the build script.
  - Con: Plugin is not visible outside the build script.
- Within a project
  - Pro: plugin is visible to every build script used by the build.
  - Con: it is not visible outside the build, and so you cannot reuse the plugin outside the build it is defined in.
- Standalone project
  - Project produces and publishes a JAR which you can then use in multiple builds and share with others.
  - Generally, this JAR might include some plugins, or bundle several related task classes into a single library.

[HOW]
Course Projects

Visual Studio Code Gradle language extension

Han Yin

- Syntax highlighting
- Keyword auto-complete proposal
- Duplication validation

Challenges

- Grammatical features of *Gradle DSL*
  - Custom AST transformations
  - Confusing script blocks
  - Each property is created with a setter with the exact same name
- Develop a language server using TypeScript
  - Npm in Node.js, TypeScript compilers, etc.

Gradle DSL Grammar

- Custom AST transformations
  - TaskContainer

```
task ("myTask1") {
    doLast {
        println "Hello #1"
    }
}

task "myTask2" {
    doLast {
        println "Hello #2"
    }
}


task myTask3  {
    doLast {
        println "Hello #3"
    }
}

task "myTask4"(type: Copy) {
    dependsOn myTask3
    doLast {
        println "Hello #4"
    }
}

task myTask5(type: Zip, dependsOn: myTask4) {
    doLast {
        println "Hello #5"
    }
}
```

```
Task ref6 = task("myTask6")

Task ref7 = task("myTask7") {
    doLast {
        println "Hello #7"
    }
}

Task ref8 = task() {
    name = "myTask8"
    doLast {
        println "Hello #8"
    }
}
```

Gradle DSL Grammar

- Confusing script blocks
  - E.g. NamedDomainObjectContainer

```
buildTypes {
    release {
        minifyEnabled true
        proguardFiles 'proguard-rules.pro'
    }

    debug {
        applicationIdSuffix ".debug"
    }
}
```

```
NamedDomainObjectContainer<BuildType> buildTypes
```

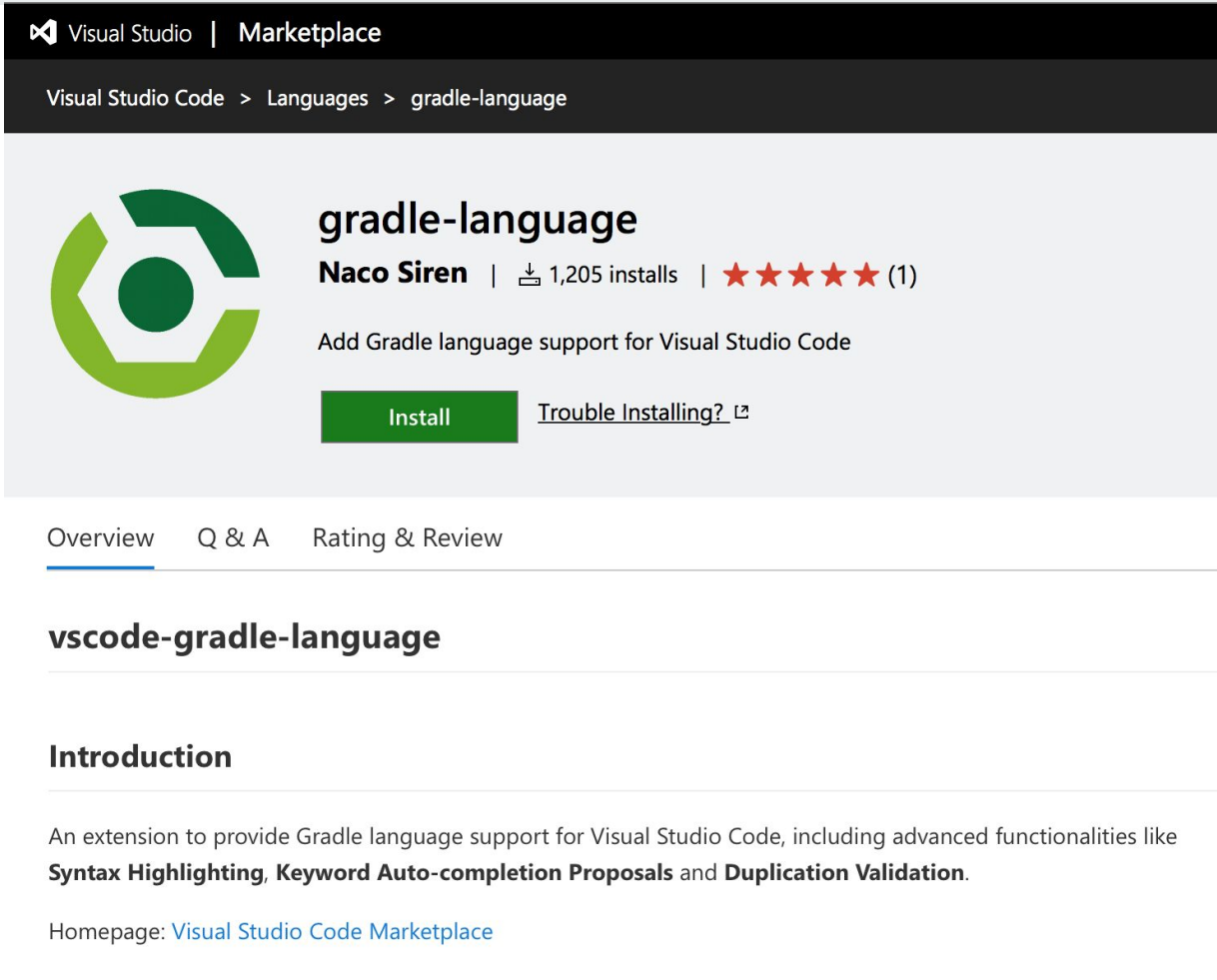
Gradle DSL Grammar

- Each property is created with a setter with the exact same name

```
1  task myTask1 {
2      description = "My 1st task"
3  }
4
5  task myTask2 {
6      description("My 2nd task")
7  }
8
9  task myTask3 {
10     description "My 3rd task"
11 }
```

Check it out!

[Homepage on Marketplace]



STARTS plugin

Hill, Randolph William

```
apply plugin: 'java'
apply plugin: 'edu.illinois.starts'

buildscript {
    repositories {
       mavenCentral()
       mavenLocal()
    }

    dependencies {
      classpath
"edu.illinois:starts-maven-plugin:1.4-
SNAPSHOT"
    }
}
```

STARTS plugin

- **Video:**

https://www.dropbox.com/s/f7g70tcys5zrm26/Starts_Update.mov?dl=0

Starts Plugin Goals

- Create corresponding ***Gradle*** tasks for each ***Maven*** goals
- Evaluate ***STARTS*** on gradle specific project and compare with ***Ekstazi***

```
# Remove all STARTS artifacts
./gradlew startsClean

# Changes since last time STARTS was run
./gradlew startsDiff

# Types impacted by changes
./gradlew startsImpacted

# Tests affected by the most recent changes
./gradlew startsSelect

# To perform RTS using STARTS
./gradlew starts
```

Docker Testing plugin

Ramey, Lloyd Nelson

- Run tests against production-like systems using docker

Docker Testing plugin

- **Video:**

https://drive.google.com/file/d/1flSwgmg6bu1MM_TuVlPNL-Efx3rek20_/view

Conclusion

Conclusion

- *Gradle* is a **build automation system** to configure, extend and customize the build process mainly for JVM languages.
- *Gradle* combines *Ant*'s power and flexibility with *Maven*'s life-cycle and ease of use by adopting a ***Groovy-based DSL*** for writing **declarative build** scripts with better concision and readability.
- *Gradle* provides outstanding extensibility and customizability by supporting various **plugins**.

Thank you!