An Introduction to Gradle for Java Developers

Kostas Saidis

www.niovity.com

Java Hellenic User Group Meetup
April 2, 2016

About Me

Kostas Saidis
saiko@niovity.com

Software & Data Architect

Twitter:
@saikos
Linkedin:
http://gr.linkedin.com/in/saiko

Academia

- BSc @ cs.unipi.gr (2001)
- MSc @ di.uoa.gr (2004)
- PhD @ di.uoa.gr (2011)

Industry

- Freelance consultant, developer & instructor since 1999
- Enterpreneur since 2011 (niovity)

Java

- Early Java enthusiast (1997)
- Groovyist since 2011

Using Gradle

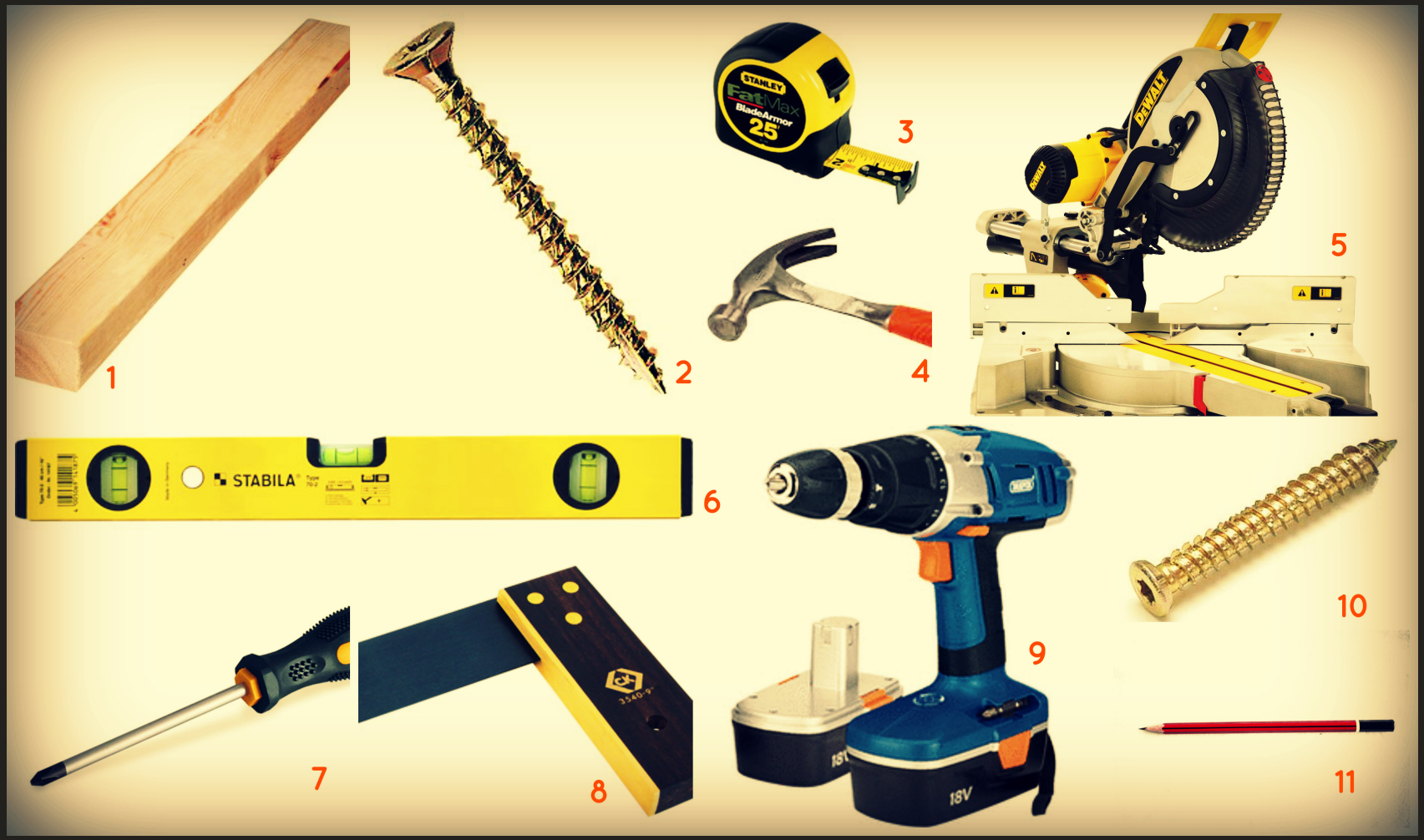
Building software with Gradle

- Since version 0.9
- Fifteen real-life projects with team sizes ranging from 2 to 10 members
- Custom plugin authoring

Target audience

Java developers
with little or no experience
in Gradle or another build tool.

Build tools



www.ayorkshirehome.com

Software build tools

Tools for managing and automating the software build process.

Software what?

Building software sucks!
(big time)

Software build process

- Develop
- Test
- Assemble
- Deploy
- Integrate

Repeat

- Again and again and again

What is Gradle?

Gradle is one of the finest software build tools out there

Makes the software build process easier to setup, extend and maintain

We focus on understanding Gradle

Gradle is a superb piece of software

What makes it so great?

- No flame wars
- No feature comparison with other tools

At a glance

Gradle

- Open source: Apache License v2
- First release 2009, latest release March 2016 (2.12)
- Backed by Gradleware... ooops, I mean Gradle Inc
- Used by: Android, Spring IO, Linkedin, Netflix, Twitter and more...

At a glance

Gradle

- Combines the best of Ant, Ivy and Maven
  - Cross-platform file management
  - Dependency management
  - Conventions
- In a smart and extensible way
  - Deep API
  - Groovy DSL
  - Easy-to-write plugins
- Ultimately offering
  - Clean and elegant builds
  - Plenty of new features
  - Better build management

Features

Non exhaustive list

- Full-fledged programmability
- Both declarative and imperative
- Convention over configuration
- Transitive dependency management
- Multi-project builds
- Polyglot (not only Java)
- Numerous plugins and integrations
- Incremental and parallel execution
- Reporting and analytics
- Embeddable
- Great documentation

The most distinct features

IMHO

1. Its superb design and technical sophistication.
2. The exposure and manipulation of its “internal” Java APIs through Groovy (aka the Groovy DSL).

A simple example

A simple Java project

```
1  apply plugin: "java"
2  group   = "org.foo.something"
3  version = "1.0-SNAPSHOT"
4  repositories {
5      mavenCentral()
6  }
7  dependencies {
8      compile     "commons-io:commons-io:2.4"
9      testCompile "junit:junit:4.11"
10     runtime     files("lib/foo.jar", "lib/bar.jar")
11 }
```

The contents of build.gradle (aka the build script), placed in the root folder of the project.

Core Concepts

Build script: a build configuration script supporting one or more projects.

Project: a component that needs to be built. It is made up of one or more tasks.

Task: a distinct step required to perform the build. Each task/step is atomic (either succeeds or fails).

Publication: the artifact produced by the build process.

Dependency Resolution

Dependencies: tasks and projects depending on each other (internal) or on third-party artifacts (external).

Transitive dependencies: the dependencies of a project may themselves have dependencies.

Repositories: the "places" that hold external dependencies (Maven/Ivy repos, local folders).

DAG: the directed acyclic graph of dependencies (what depends on what).

Dependency configurations : named sets (groups) of dependencies (e.g. per task).

Plugins

A plugin applies a set of extensions to the build process.

- Add taks to a project.
- Pre-configure these tasks with reasonable defaults.
- Add dependency configurations.
- Add new properties and methods to existing objects.

Plugins implement the “build-by-convention” principle in a flexible way.

The Build Lifecycle

1. Initialization: initialization of the project.
2. Configuration: configuration of the project (computes the DAG).
3. Execution: executes the sequence of build tasks.

The simple example

```
1  apply plugin: "java"
2  //Introduces a set of Maven-style conventions
3  //(tasks, source locations, dependency configurations, etc)
4  group   = "org.foo.something"
5  version = "1.0-SNAPSHOT"
6  repositories {
7  //resolve all external dependencies via Maven central
8      mavenCentral()
9  }
10 dependencies {
11     //each name (compile, testCompile, etc) refers to
12     //a configuration introduced by the java plugin
13     compile     "commons-io:commons-io:2.4"
14     testCompile "junit:junit:4.11"
15     runtime     files("lib/foo.jar", "lib/bar.jar")
16 }
```

Invoking gradle

Run a build task

```
> gradle test
```

Compiles the sources and runs the tests

```
> gradle tasks
```

clean, assemble, build, classes, testClasses, test, jar, etc

Never invoke gradle directly but always use the gradle wrapper: a mechanism to execute gradle without manually installing it beforehand (not covered here).

A more complex example

Ant integration, custom tasks

```
1  apply plugin:'java'
2  task generateFiles(type: JavaExec) {
3      main = 'some.class.name'
4      classpath = sourceSets.main.runtimeClasspath
5      args = [ projectDir, 'path/to/gen/files' ]
6  }
7  test {
8      dependsOn generateFiles
9      doLast {
10         ant.copy(toDir:'build/test-classes'){
11             fileset dir:'path/to/gen/files'
12         }
13     }
14 }
15 import org.apache.commons.io.FileUtils
16 clean.doFirst {
17     FileUtils.deleteQuietly(new File('path/to/gen/files'))
18 }
```

Hmmm

What's going on here?

Let's have a closer look

```
1  //apply the java plugin as before
2  apply plugin:'java'
3  //introduce a new task (that invokes a java process)
4  task generateFiles(type: JavaExec) {
5      main = 'some.class.name'
6      classpath = sourceSets.main.runtimeClasspath
7      args = [ projectDir, 'path/to/gen/files' ]
8  }
9  //customize the test task
10 test {
11     dependsOn generateFiles
12     doLast {
13         ant.copy(toDir:'build/test-classes'){
14             fileset dir:'path/to/gen/files'
15         }
16     }
17 }
18 //import a class required by this build script
19 //some necessary classpath definitions are not shown for brevity
20 import org.apache.commons.io.FileUtils
21 //customize the clean task
22 clean.doFirst {
23     FileUtils.deleteQuietly(new File('path/to/gen/files'))
24 }
```

What's that strange syntax?

It's Groovy!

groovy-lang.org

Gradle's power lies in its Groovy-based DSL

What you need to know about Groovy to start using Gradle in its full potential

For more Groovy magic, have a look at:

An Introduction to Groovy for Java Developers

What is Groovy?

Groovy is a feature-rich, Java-friendly language for the JVM

- A super version of Java
  - Augments Java with additional features
- Designed as a companion language for Java
  - seamless integration with Java $\rightarrow$ an additional jar at runtime!
  - syntactically aligned with Java $\rightarrow$ syntactically correct Java will work in Groovy (with some gotchas)!
  - compiles into JVM bytecode and preserves Java semantics $\rightarrow$ call Groovy from Java == call Java from Java!
- the 2nd language targeting the Java platform (JSR-241) $\rightarrow$ Java was the first!

Groovy Scripts

Java

```
1  public class Booh {
2      public static void main(String[] args) {
3          System.out.println("Booh booh");
4      }
5  }
```

Compile and run it.

Groovy

```
1  println("Booh booh")
```

Run it directy.

POJOs vs POGOs

Java

```
1  public class Foo {
2      private String message;
3      public void setMessage(String message) {
4          this.message = message;
5      }
6      public String getMessage() {
7          return message;
8      }
9  }
```

Groovy

```
1  class Foo {
2      String message
3  }
```

Groovy Map Syntax

Java

```
1 Map<String, String> pairs = new HashMap();
2 pairs.put("one", "1");
3 pairs.put("two", "2");
```

Groovy

```
1 Map<String, String> pairs = [one:'1', two:"2"]
```

Groovy Operator Overloading

```
1  import java.util.concurrent.*
2  class Tasks {
3      List<Callable> taskList = []
4      void execute() {
5          for(Callable t: taskList) {
6              t.run()
7          }
8      }
9      void leftShift(Tasks tasks) {
10         taskList.addAll(tasks.taskList)
11     }
12 }
13 Tasks t1 = new Tasks()
14 ...
15 Tasks t2 = new Tasks()
16 ...
17 t1 << t2
```

Groovy Closures

A closure is an anonymous function together with a referencing environment.

- may accept parameters or return a value,
- can be assigned to variables,
- can be passed as argument,
- captures the variables of its surrounding lexical scope.

Example

```
1  class Test {
2      long x = 3
3      //a block of code assigned to a variable
4      Closure<Long> times = { long l -> l * x }
5      //x is the free variable
6  }
7  Test test = new Test()
8  assert test.times(3) == 9
9  test.x = 4
10 assert test.times(3) == 12
11 class Test2 {
12     long x = 0
13 }
14 test.times.resolveStrategy = Closure.DELEGATE_FIRST
15 test.times.delegate = new Test2()
16 assert test.times(3) == 0
```

Groovy method invocation syntax

```groovy
//omit parentheses
println("booh booh")
println "booh booh"
method "one", 2
//omit map's braces
method([one:1, two: 2])
method(one:1, two:2)
method one:1, two:2
//The closure as last argument pattern
Closure clos = { int x, int y -> x + y }
method("one", 2, clos)
method "one", 2, clos
method("one", 2) { int x, int y ->
    x + y
}
```

Behind the scenes

- Gradle build scripts are Groovy scripts.
- Such Groovy scripts operate in the context of Gradle's domain objects (DSL objects).
- The main DSL object is a `org.gradle.api.Project` instance.
- There is one-to-one relationship between a Projet object and a build.gradle file.
- Each DSL object exposes its own properties and methods.

Have a look at: Gradle DSL Reference

Putting it all together

Epiphany

```
1  //a groovy script with a Project object as the context
2  apply plugin: "java"
3  //invocation of Project.apply(Map) method
4  group   = "org.foo.something"
5  version = "1.0-SNAPSHOT"
6  //update of project.group, project.version properties
7  repositories {
8      mavenCentral()
9  }
10 //invocation of the Project.repositories(Closure) method
11 //From the docs: the closure has a RepositoryHandler object
12 //as its delegate
13 dependencies {
14     compile     "commons-io:commons-io:2.4"
15     testCompile "junit:junit:4.11"
16     runtime     files("lib/foo.jar", "lib/bar.jar")
17 }
18 //here?
```

And here?

```
apply plugin:'java'
task generateFiles(type: JavaExec) {
    main = 'some.class.name'
    classpath = sourceSets.main.runtimeClasspath
    args = [ projectDir, 'path/to/gen/files' ]
}
test {
    dependsOn generateFiles
    doLast {
        ant.copy(toDir:'build/test-classes'){
            fileset dir:'path/to/gen/files'
        }
    }
}
import org.apache.commons.io.FileUtils
clean.doFirst {
    FileUtils.deleteQuietly(new File('path/to/gen/files'))
}
```

Configuration vs Execution Phase

```
1  test {//closure runs at configuration phase
2      dependsOn generateFiles
3      doLast { //at execution phase
4          ant.copy(toDir:'build/test-classes'){
5              fileset dir:'path/to/gen/files'
6          }
7      }
8  }
9  test.doLast { //at execution phase
10     ant.copy...
11 }
12 task test << { //at execution phase
13     ant.copy...
14 }
15 test { //at configuration phase
16     ant.copy...
17 }
18 task test { //at configuration phase
19     ant.copy...
20 }
```

Questions?

Proudly powered by:
LaTeX

Using the Beamer class &
the Wronki theme (slightly modified).