Gradle

Gradle is an open-source build automation tool focused on flexibility and performance. Gradle build scripts are written using a Groovy domain specific language (DSL).

The idea behind **build automation** is to increase software developer's productivity by automating manual tasks. Think, for example, when you write a Java program as an assignment: the first step is to setup your base code (code provided for the worksheet), then you have to write your program. After that, you have to compile it and run it by using the corresponding Java tools (javac/java). Then you would normally desk check that the program behaves as expected. When you are satisfied with your solution you have to copy your file in a folder (to keep a backup) and then the file has to be submitted. Well, most of those steps could automated using a Gradle build script allowing you to submit every time the program compiles and runs successfully. If, in addition, a test suite is included to check the correctness of the program. Every time you compile the program, it would be checked for correctness and, then, it would be backed up and submitted - automatically. For such a small example, a build script could save you up to several minutes. Imagine the savings when a system becomes larger and involves several developers.

Among Gradle's features, we are going to focus on **dependency management** and some of the others will be mentioned as they are used. In this first session, the Gradle tooling is introduced and the basic building blocks of a Gradle build script are explained: projects, tasks and task dependencies.

Outline of Concepts

Everything in Gradle sits on top of two basic concepts: projects and tasks.

Every Gradle build is made up of one or more projects. What a **project** represents depends on what it is that you are doing with Gradle. For example, a project might represent a library JAR or a web application. It might represent a distribution ZIP assembled from the JARs produced by other projects. A project does not necessarily represent a thing to be built. It might represent a thing to be done, such as deploying your application to staging or production environments. Don't worry if this seems a little vague for now. Gradle's build-by-convention support adds a more concrete definition for what a project is.

A **build** consists of one or more projects.

Each project is made up of one or more tasks. 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. When a task `B` requires the execution of a task `A`, we say that task `B` depends on task `A` or that task `A` is a **dependency** for task `B`.

In the following we will learn how to use Gradle and then how to declare tasks and dependencies among them.

Running Gradle from a terminal

The contents in this section are extracted from the :books: documentation on Command-Line Interface, which provides a more comprehensive coverage of the `gradle` command.

The gradle command

From terminal, gradle can be executed using the following command: `gradle [taskName...] [--option-name...]`

Options are allowed before and after task names.

If multiple tasks are specified, they should be separated with a space.

Options that accept values can be specified with or without `=` between the option and argument; however, use of `=` is recommended.

```
--console=plain
```

Options that enable behaviour have long-form options with inverses specified with `--no-`. The following are opposites.

```
--build-cache
--no-build-cache
```

Many long-form options, have short option equivalents. The following are equivalent:

```
--help
-h
```

Many command-line flags can be specified in gradle.properties to avoid needing to be typed. See the configuring build environment guide for details.

Executing tasks

In a terminal, change directory (`cd`) to the root folder of your project - or where the `build.gradle` file is. There you can run a task `TASK_NAME` and all of its dependencies using the following command:

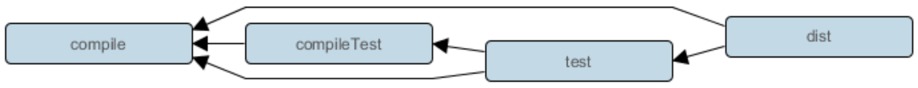
```
gradle TASK_NAME
```

:loudspeaker: To simplify the introduction to Gradle, feel free to **skip any documentation regarding multi-projects** in CO2006.

To execute multiple tasks, e.g. `taskA` and `taskB` use:

```
gradle taskA taskB
```

You can exclude a task from being executed using the `-x` or `--exclude-task` command-line option and providing the name of the task to exclude.



For example if we execute `gradle dist --exclude-task test` we obtain the following output:

```
> gradle dist --exclude-task test

> Task :compile
compiling source

> Task :dist
building the distribution

BUILD SUCCESSFUL in 0s
2 actionable tasks: 2 executed
```

The `test` task is not executed, even though it is a dependency of the `dist` task. The `test` task's dependencies such as `compileTest` are not executed either. Those dependencies of `test` that are required by another task, such as compile, are still executed.

Common tasks

The following are task conventions applied by built-in and most major Gradle plugins.

- Computing all outputs: it is common in Gradle builds for the build task to designate assembling all outputs and running all checks.

```
gradle build
```

- Running applications: it is common for applications to be run with the run task, which assembles the application and executes some script or binary.

```
gradle run
```

- Running all checks: it is common for all verification tasks, including tests and linting, to be executed using the check task.

```
gradle check
```

- Cleaning outputs: the contents of the build directory can be deleted using the clean task, though doing so will cause pre-computed outputs to be lost, causing significant additional build time for the subsequent task execution.

```
gradle clean
```

Project reporting

Gradle provides several built-in tasks which show particular details of your build. This can be useful for understanding the structure and dependencies of your build, and for debugging problems.

You can get basic help about available reporting options using `gradle help`.

- Listing projects: running `gradle projects` gives you a list of the subprojects of the selected project, displayed in a hierarchy.
- Listing tasks: running `gradle tasks` gives you a list of the main tasks of the selected project. This report shows the default tasks for the project, if any, and a description for each task. By default, this report shows only those tasks which have been assigned to a task group. You can obtain more information in the task listing using the –all option.

```
gradle tasks --all
```

- Show task usage details: running `gradle help --task someTask` gives you detailed information about a specific task.
- Listing project dependencies: running `gradle dependencies` gives you a list of the dependencies of the selected project, broken down by configuration. For each configuration, the direct and transitive dependencies of that configuration are shown in a tree. Concrete examples of build scripts and output available in the documentation on Inspecting Dependencies.
- Listing project properties: running `gradle properties` gives you a list of the properties of the selected project.

Running Gradle from STS (Eclipse)

You can also run Gradle from the Spring Tool Suite (Eclipse) with Gradle Buildship, using view `Gradle tasks`. Information about how to do it is available in :books: this tutorial (sections 3-5).

:loudspeaker: Note that this software **is already installed** in the Linux machines in the lab.

Projects and tasks

The contents in this section are extracted from the :books: documentation on Build Script Basics.

Project object

A Project object is created every time you run a Gradle build script. This object provides some standard properties (e.g. the name of the project), which are available in your build script. The following table lists a few of the commonly used ones. See this table for an outline of them.

Defining tasks

You run a Gradle build using the gradle command. The gradle command looks for a file called `build.gradle` in the current directory. We call this `build.gradle` file a **build script**, although strictly speaking it is a build configuration script. The build script defines a project and its tasks.

- A task `TaskA` can be defined as follows:

```
task TaskA
TaskA.description = "task A"
```

- To add actions to a task use the following:

```
TaskA.doLast { println "task A" }
TaskA << { println "task A" }
TaskA.doFirst { println "at the start of task A" }
```

Remember: **build scripts are code** and Gradle's build scripts give you the full power of Groovy.

- Tasks can be written as closures:

```
task TaskA {
  description = "task A"
  doLast {
    println "taskA"
  }
}
```

To execute a task use the gradle command from a terminal or from the Gradle task view in the STS as explained in the section above. For example, in a terminal, use `gradle -q TaskA` (where `-q` suppresses Gradle's log messages).

Defining task dependencies

You can declare tasks that depend on other tasks:

- `TaskA` can execute only if `TaskB` is executed:

```
task TaskA
task TaskB TaskA.dependsOn TaskB
```

- equivalently (enforces predecessor):

```
task TaskA {
    dependsOn TaskB
}
task TaskB
```

- similarly (enforces successor):

```
task TaskA
task TaskB
TaskB.finalizedBy TaskA
```

More information on adding dependencies to a task can be found here.

User-defined properties

All enhanced objects in Gradle's domain model can hold extra user-defined properties. This includes, but is not limited to, projects, tasks, and source sets. Extra properties can be added, read and set via the owning object's `ext` property. Alternatively, an `ext` block can be used to add multiple properties at once.

Variables can be used to declare properties and these can be: * local to a method or to a project as in Groovy:

```
def version = "1.0"
task TaskA {
    description = "task A - version $version"
}
```

- global using the `ext` property of the `project` object:

```
project.ext.version = "1.0"
task TaskA {
  description = "task A - version $version"
}
```

or

```
project {
  ext {
    version = "1.0"
  }
}
```

Typed tasks

Gradle supports enhanced tasks, which are tasks that have their own properties and methods. The tasks that are predefined and can be reused are documented here.

For example, we can declare a Copy task using the following syntax:

```
task(copyFiles, type: Copy) {
    from(file('srcDir'))
    into(buildDir)
}
```

or

```
task copyFiles(type: Copy) {
    from 'source '
    into 'target'
}
```

The name of this task is `copyFiles`, but it is of type `Copy`. You can have multiple tasks of the same type, but with different names.

See :books: this section for more examples on how to configure the Copy task.

References (and further reading)

- Getting started: official Gradle tutorial to get started with Gradle.
- Using Gradle
- Command line interface: the gradle command.
- Eclipse's Gradle Task view (sections 1-5): executing Gradle from Eclipse (i.e. from the Spring Tool Suite - STS)
- Building scripts basics: more information on tasks
- Writing build scripts: using the `Project` API for configuring scripts

References

References