

Gradle

A build system

Version 0.8

Hans Dockter

Adam Murdoch

Copyright © 2007-2009 Hans Dockter, Adam Murdoch

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

Table of Contents

1. Introduction	
1.1. About this user guide	
2. Overview	
2.1. Features	
2.2. Why Groovy?	
2.3. Missing features	
3. Getting Started	
3.1. Prerequisites	
3.2. Download	
3.3. Unpacking	
3.4. Environment variables	
3.5. Running and testing your installation	
3.6. JVM options	
4. Build Script Basics	
4.1. Hello world	
4.2. Build scripts are code	
4.3. Task dependencies	
4.4. Dynamic tasks	
4.5. Manipulating existing tasks	
4.6. Shortcut notations	
4.7. Dynamic task properties	
4.8. Using Ant Tasks	
4.9. Using methods	
4.10. Default tasks	
4.11. Configure by DAG	
4.12. Summary	
5. Java Quickstart	
5.1. A basic Java project	
5.2. Multi-project Java build	
5.3. Summary	
6. Groovy Quickstart	
6.1. A basic Groovy project	
6.2. Summary	
7. Web Application Quickstart	
7.1. Building a WAR file	
7.2. Running your web application	
7.3. Summary	
8. Artifact Basics	
8.1. Artifact configurations	
8.2. Repositories	
8.3. External dependencies	
8.4. Artifact publishing	
8.5. API	
9. Using the Gradle Command-Line	
9.1. Executing multiple tasks	
9.2. Excluding tasks	
9.3. Task name abbreviation	
9.4. Selecting which build to execute	
9.5. Obtaining information about your build	
9.6. Dry Run	
9.7. Summary	
10. Using the Gradle Graphical User Interface	

- 10.1. Task Tree
- 10.2. Favorites
- 10.3. Command Line
- 10.4. Setup

11. Tutorial - 'This and That'

- 11.1. Directory creation
- 11.2. Gradle properties and system properties
- 11.3. Accessing the web via a proxy
- 11.4. Caching
- 11.5. Configuring arbitrary objects

12. The Project and Task API

- 12.1. Project API
- 12.2. Task API
- 12.3. Summary

13. More about Tasks

- 13.1. Defining tasks
- 13.2. Locating tasks
- 13.3. Configuring tasks
- 13.4. Adding dependencies to a task
- 13.5. Adding a description to a task
- 13.6. Replacing tasks
- 13.7. Skipping tasks
- 13.8. Task rules
- 13.9. Summary

14. Working With Files

- 14.1. Locating files
- 14.2. File collections
- 14.3. File trees
- 14.4. Specifying a set of files
- 14.5. Copying files

15. Logging

- 15.1. Choosing a log level
- 15.2. Writing your own log messages
- 15.3. Logging from external tools and libraries

16. Using Ant from Gradle

- 16.1. Using Ant tasks and types in your build
- 16.2. Importing an Ant build
- 16.3. Ant properties and references
- 16.4. API

17. Using Plugins

- 17.1. Declaring plugins
- 17.2. Using the convention object
- 17.3. Summary

18. The Java Plugin

- 18.1. Source sets
- 18.2. Tasks
- 18.3. Project layout
- 18.4. Dependency management
- 18.5. Convention properties
- 18.6. Working with source sets
- 18.7. Javadoc
- 18.8. Clean
- 18.9. Resources
- 18.10. CompileJava
- 18.11. Test
- 18.12. Jar

- 18.13. Adding archives
- 18.14. Uploading
- 18.15. Eclipse

19. The Groovy Plugin

- 19.1. Tasks
- 19.2. Project layout
- 19.3. Dependency management
- 19.4. Convention properties
- 19.5. Source set properties
- 19.6. CompileGroovy

20. The Scala Plugin

- 20.1. Tasks
- 20.2. Project layout
- 20.3. Dependency Management
- 20.4. Convention Properties
- 20.5. Source set properties

21. The War Plugin

- 21.1. Tasks
- 21.2. Project layout
- 21.3. Dependency management
- 21.4. Convention properties
- 21.5. War
- 21.6. Customizing
- 21.7. Eclipse WTP

22. The Jetty Plugin

- 22.1. Tasks
- 22.2. Project layout
- 22.3. Dependency management
- 22.4. Convention properties

23. The Maven Plugin

- 23.1. Tasks
- 23.2. Project layout
- 23.3. Dependency management
- 23.4. Convention properties

24. The Code Quality Plugin

- 24.1. Tasks
- 24.2. Project layout
- 24.3. Dependency management
- 24.4. Convention properties

25. The OSGi Plugin

- 25.1. Tasks
- 25.2. Project layout
- 25.3. Dependency management
- 25.4. Convention properties

26. The Eclipse Plugin

- 26.1. Usage
- 26.2. Tasks

27. The Project Report Plugin

- 27.1. Tasks
- 27.2. Project layout
- 27.3. Dependency management
- 27.4. Convention properties

28. Dependency Management

- 28.1. Introduction
- 28.2. Dependency management overview

28.3. How to declare your dependencies	
28.4. Working with dependencies	
28.5. Repositories	
28.6. Strategies for transitive dependency management	
29. Artifact Management	
29.1. Introduction	
29.2. Artifacts and configurations	
29.3. Uploading artifacts	
29.4. More about project libraries	
29.5. Interacting with Maven repositories	
30. The Build Lifecycle	
30.1. Build phases	
30.2. Settings file	
30.3. Multi-project builds	
30.4. Initialization	
30.5. Configuration and execution of a single project build	
30.6. Responding to the lifecycle in the build script	
31. Multi-project Builds	
31.1. Cross project configuration	
31.2. Subproject configuration	
31.3. Execution rules for multi-project builds	
31.4. Running tasks by their absolute path	
31.5. Project and task paths	
31.6. Dependencies - Which dependencies?	
31.7. Project lib dependencies	
31.8. Multi-Project Building and Testing	
31.9. Property and method inheritance	
31.10. Summary	
32. Writing Custom Tasks	
33. Writing Custom Plugins	
34. Organizing Build Logic	
34.1. Build sources	
34.2. External dependencies for the build script	
34.3. Ant optional dependencies	
34.4. Summary	
35. Initialization Scripts	
35.1. Basic usage	
35.2. External dependencies for the init script	
36. The Gradle Wrapper	
36.1. Configuration	
36.2. Unix file permissions	
36.3. Environment variable	
37. Embedding Gradle	
A. Potential Traps	
A.1. Groovy script variables	
A.2. Configuration and execution phase	
B. Gradle Command Line	
C. Existing IDE Support and how to cope without it	
C.1. IntelliJ	
C.2. Eclipse	
C.3. Using Gradle without IDE support	

List of Examples

- 4.1. The first build script
- 4.2. Execution of a build script
- 4.3. Using Groovy in Gradle's tasks
- 4.4. Using Groovy in Gradle's tasks
- 4.5. Declaration of dependencies between tasks
- 4.6. Lazy dependsOn - the other task does not exist (yet)
- 4.7. Dynamic creation of a task
- 4.8. Accessing a task via API - adding a dependency
- 4.9. Accessing a task via API - adding behaviour
- 4.10. Accessing task as a property of the build script
- 4.11. Assigning properties to a task
- 4.12. Using AntBuilder to execute ant.checksum target
- 4.13. Using methods to organize your build logic
- 4.14. Defining a default tasks
- 4.15. Different outcomes of build depending on chosen tasks
- 5.1. Using the Java plugin
- 5.2. Building a Java project
- 5.3. Adding Maven repository
- 5.4. Adding dependencies
- 5.5. Customization of MANIFEST.MF
- 5.6. Adding system property
- 5.7. Publishing the JAR file
- 5.8. Eclipse plugin
- 5.9. Java example - complete build file
- 5.10. Multi-project build - hierarchical layout
- 5.11. Multi-project build - settings.gradle file
- 5.12. Multi-project build - common configuration
- 5.13. Multi-project build - dependencies between projects
- 5.14. Multi-project build - distribution file
- 6.1. Groovy plugin
- 6.2. Dependency on Groovy 1.6.0
- 6.3. Groovy example - complete build file
- 7.1. War plugin
- 7.2. Running web application with Jetty plugin
- 8.1. Definition of a configuration
- 8.2. Accessing a configuration
- 8.3. Configuration of a configuration
- 8.4. Usage of Maven central repository
- 8.5. Usage of a local directory
- 8.6. Accessing a repository
- 8.7. Configuration of a repository
- 8.8. Definition of an external dependency
- 8.9. Usage of external dependency of a configuration
- 9.1. Executing multiple tasks
- 9.2. Excluding tasks
- 9.3. Abbreviated task name
- 9.4. Abbreviated camel case task name
- 9.5. Selecting the project using a build file

- 9.6. [Selecting the project using project directory](#)
- 9.7. [Obtaining information about tasks](#)
- 9.8. [Obtaining information about dependencies](#)
- 10.1. [Launching the GUI](#)
- 11.1. [Directory creation with mkdir](#)
- 11.2. [Directory creation with Directory tasks](#)
- 11.3. [Setting properties with a gradle.properties file](#)
- 11.4. [Accessing the web via a proxy](#)
- 11.5. [Configuring arbitrary objects](#)
- 12.1. [Accessing property of the Project object](#)
- 12.2. [Project properties](#)
- 13.1. [Defining tasks](#)
- 13.2. [Defining tasks - using strings](#)
- 13.3. [Defining tasks with alternative syntax](#)
- 13.4. [Accessing tasks as properties](#)
- 13.5. [Accessing tasks via tasks collection](#)
- 13.6. [Accessing tasks by path](#)
- 13.7. [Creating a copy task](#)
- 13.8. [Configuring a task - various ways](#)
- 13.9. [Configuring a task - fluent interface](#)
- 13.10. [Configuring a task - with closure](#)
- 13.11. [Configuring a task - with configure\(\) method](#)
- 13.12. [Defining a task with closure](#)
- 13.13. [Adding dependency on task from another project](#)
- 13.14. [Adding dependency using task object](#)
- 13.15. [Adding dependency using closure](#)
- 13.16. [Adding a description to a task](#)
- 13.17. [Overwriting a task](#)
- 13.18. [Skipping a task using a predicate](#)
- 13.19. [Skipping tasks with StopExecutionException](#)
- 13.20. [Enabling and disabling tasks](#)
- 13.21. [Task rule](#)
- 13.22. [Dependency on rule based tasks](#)
- 14.1. [Locating files](#)
- 14.2. [Creating a file collection](#)
- 14.3. [Using a file collection](#)
- 14.4. [Implementing a file collection](#)
- 14.5. [Creating a file tree](#)
- 14.6. [Using a file tree](#)
- 14.7. [Specifying a set of files](#)
- 14.8. [Specifying a set of files](#)
- 14.9. [Copying files using the copy task](#)
- 14.10. [Copying files using the copy\(\) method](#)
- 15.1. [Using stdout to write log messages](#)
- 15.2. [Writing your own log messages](#)
- 15.3. [Using SLF4J to write log messages](#)
- 15.4. [Configuring standard output capture](#)
- 15.5. [Configuring standard output capture for a task](#)

- 16.1. [Using an Ant task](#)
- 16.2. [Passing nested text to an Ant task](#)
- 16.3. [Passing nested elements to an Ant task](#)
- 16.4. [Using an Ant type](#)
- 16.5. [Using a custom Ant task](#)
- 16.6. [Declaring the classpath for a custom Ant task](#)
- 16.7. [Using a custom Ant task and dependency management together](#)
- 16.8. [Importing an Ant build](#)
- 16.9. [Task that depends on Ant target](#)
- 16.10. [Adding behaviour to an Ant target](#)
- 16.11. [Ant target that depends on Gradle task](#)
- 16.12. [Setting an Ant property](#)
- 16.13. [Getting an Ant property](#)
- 16.14. [Setting an Ant reference](#)
- 16.15. [Getting an Ant reference](#)
- 17.1. [Using plugin](#)
- 17.2. [Configuring a plugin task](#)
- 17.3. [Plugin convention object](#)
- 17.4. [Using the plugin convention object](#)
- 18.1. [Custom Java source layout](#)
- 18.2. [Accessing a source set](#)
- 18.3. [Configuring the source directories of a source set](#)
- 18.4. [Defining a source set](#)
- 18.5. [Defining the classpath of a source set](#)
- 18.6. [Assembling a JAR for a source set](#)
- 18.7. [Generating the Javadoc for a source set](#)
- 18.8. [Running tests in a source set](#)
- 18.9. [Creation of ZIP archive](#)
- 18.10. [Configuration of archive task - custom archive name](#)
- 18.11. [Configuration of archive task - appendix & classifier](#)
- 18.12. [Adding content to archive - include & exclude](#)
- 18.13. [Adding content to archive - arbitrary files](#)
- 18.14. [Adding content to archive - zipFileSet](#)
- 18.15. [Creation of TAR archive](#)
- 18.16. [Customization of MANIFEST.MF](#)
- 18.17. [Customization of MANIFEST.MF for a particular archive](#)
- 19.1. [Custom Groovy source layout](#)
- 19.2. [Configuration of Groovy plugin](#)
- 19.3. [Configuration of Groovy plugin](#)
- 20.1. [Custom Scala source layout](#)
- 20.2. [Declaring the Scala version to use](#)
- 21.1. [Customization of war plugin](#)
- 21.2. [Generation of JAR archive in addition to WAR archive](#)
- 25.1. [Configuration of OSGi MANIFEST.MF file](#)
- 28.1. [Module dependencies](#)
- 28.2. [Artifact only notation](#)
- 28.3. [Dependency with classifier](#)
- 28.4. [Client module dependencies - transitive dependencies](#)

- 28.5. [Project dependencies](#)
- 28.6. [File dependencies](#)
- 28.7. [Generated file dependencies](#)
- 28.8. [Excluding transitive dependencies](#)
- 28.9. [Optional attributes of dependencies](#)
- 28.10. [Collections and arrays of dependencies](#)
- 28.11. [Dependency configurations](#)
- 28.12. [Dependency configurations for project](#)
- 28.13. [Configuration.copy](#)
- 28.14. [Accessing declared dependencies](#)
- 28.15. [Configuration.files](#)
- 28.16. [Configuration.files with spec](#)
- 28.17. [Configuration.copy](#)
- 28.18. [Configuration.copy vs. Configuration.files](#)
- 28.19. [Adding central Maven repository](#)
- 28.20. [Adding many Maven repositories](#)
- 28.21. [Adding custom Maven repository](#)
- 28.22. [Adding additional Maven repositories for JAR files](#)
- 28.23. [Accessing password protected Maven repository](#)
- 28.24. [Flat repository resolver](#)
- 28.25. [Definition of a custom repository](#)
- 29.1. [Assignment of an artifact to a configuration](#)
- 29.2. [Configuration of the upload task](#)
- 29.3. [Upload of file to remote Maven repository](#)
- 29.4. [Upload of file via SSH](#)
- 29.5. [Customization of pom](#)
- 29.6. [Customization of Maven installer](#)
- 29.7. [Generation of multiple poms](#)
- 29.8. [Accessing a mapping configuration](#)
- 30.1. [Single project build](#)
- 30.2. [Hierarchical layout](#)
- 30.3. [Flat layout](#)
- 30.4. [Modification of elements of the project tree](#)
- 30.5. [Modification of elements of the project tree](#)
- 30.6. [Adding of test task to each project which has certain property set](#)
- 30.7. [Notifications](#)
- 30.8. [Setting of certain property to all tasks](#)
- 30.9. [Logging of start and end of each task execution](#)
- 31.1. [Multi-project tree - water & bluewhale projects](#)
- 31.2. [Build script of water \(parent\) project](#)
- 31.3. [Multi-project tree - water, bluewhale & krill projects](#)
- 31.4. [Water project build script](#)
- 31.5. [Defining common behaviour of all projects and subprojects](#)
- 31.6. [Defining specific behaviour for particular project](#)
- 31.7. [Defining specific behaviour for project krill](#)
- 31.8. [Adding custom behaviour to some projects \(filtered by project name\)](#)
- 31.9. [Adding custom behaviour to some projects \(filtered by project properties\)](#)
- 31.10. [Running build from subproject](#)

- 31.11. Evaluation and execution of projects
- 31.12. Evaluation and execution of projects
- 31.13. Running tasks by their absolute path
- 31.14. Dependencies and execution order
- 31.15. Dependencies and execution order
- 31.16. Dependencies and execution order
- 31.17. Declaring dependencies
- 31.18. Declaring dependencies
- 31.19. Project execution dependencies
- 31.20. Cross project task dependencies
- 31.21. Configuration time dependencies
- 31.22. Configuration time dependencies - evaluationDependsOn
- 31.23. Configuration time dependencies
- 31.24. Dependencies - real life example - crossproject configuration
- 31.25. Project lib dependencies
- 31.26. Project lib dependencies
- 31.27. Fine grained control over dependencies
- 31.28. Build and Test Single Project
- 31.29. Partial Build and Test Single Project
- 31.30. Build and Test Depended On Projects
- 31.31. Build and Test Dependent Projects
- 32.1. Defining a custom task
- 32.2. A hello world task
- 32.3. A customizable hello world task
- 33.1. A custom plugin
- 33.2. A custom plugin convention
- 34.1. Declaring external dependencies for the build script
- 34.2. A build script with external dependencies
- 34.3. Ant optional dependencies
- 35.1. Declaring external dependencies for an init script
- 35.2. An init script with external dependencies
- 36.1. Wrapper task
- 36.2. Configuration of wrapper task
- A.1. Variables scope: local and script wide
- A.2. Distinct configuration and execution phase

Introduction

We would like to introduce Gradle to you, a build system that we think is a quantum leap for build technology in the Java (JVM) world. Gradle provides:

- A very flexible general purpose build tool like Ant.
- Switchable, build-by-convention frameworks a la Maven. But we never lock you in!
- Very powerful support for multi-project builds.
- Very powerful dependency management (based on Apache Ivy).
- Full support for your existing Maven or Ivy repository infrastructure.
- Support for transitive dependency management without the need for remote repositories or `pom.xml` and `ivy.xml` files.
- Ant tasks and builds as first class citizens.
- *Groovy* build scripts.
- A rich domain model for describing your build.

In [Chapter 2, Overview](#) you will find a detailed overview of Gradle. Otherwise, the [tutorials](#) are waiting, have fun :)

1.1. About this user guide

This user guide, like Gradle itself, is under very active development. Some parts of Gradle aren't documented as completely as they need to be. Some of the content presented won't be entirely clear or will assume that you know more about Gradle than you do. We need your help to improve this user guide. You can find out more about contributing to the documentation at the [Gradle web site](#).

You can find more examples, and some additions to this user guide, on the [wiki](#). You can also contribute your own examples and extra content there.

Overview

2.1. Features

Here is a list of some of Gradle's features.

Language for Dependency Based Programming

This is the core of Gradle. Most build tools do offer such a thing. You can create tasks, create dependencies between them and those tasks get executed only once and in the right order. Yet compared to Ant ^[1] Gradle's task offer a rich API and can be any kind of object. Gradle's tasks support multi-project builds. There is much more to say about tasks later on.

Flexible Build By Convention

Gradle offers you build-by-convention *on top* of its core layer. It is the same idea as implemented by Maven. But Gradle's build-by-convention approach is highly configurable and flexible. And you don't have to use it, if you need utmost flexibility. You can enable/disable it on a per project basis in a multi-project build.

Ant Tasks

Ant tasks are first class citizens. Using Ant tasks from Gradle is as convenient and more powerful than using Ant tasks from a `build.xml` file.

Configure By DAG

Gradle has a distinct configuration and execution phase. Thus we can offer you special hooks. You can add configuration to your build, based on the complete execution graph of tasks, before any task is executed.

Easy Ivy

Our dependency management is based on Apache Ivy, the most advanced and powerful dependency management in the Java world. We have Ivy integrated in our build-by-convention framework. It is ready to go out-of-the-box. Ivy is mostly used via its Ant tasks but it also provides an API. Gradle integrates deeply with Ivy via this API. Gradle has its own dependency DSL on top of Ivy. This DSL introduces a couple of features not provided by Ivy itself.

Client Modules

We think dependency management is important to any project. *Client Modules* provide this, without the need of remote repositories and `ivy.xml` or `pom.xml` files. For example you can just put your jars into svn and yet enjoy complete transitive dependency management. Gradle also support fully Ivy or Maven repository infrastructures based on `ivy.xml` or `pom.xml` files and remote repositories.

Cross Project Configuration

Enjoy how easy and yet how extremely powerful the handling of multi-project builds can be. Gradle introduces *Configuration Injection* to make this possible.

Distinct Dependency Hierarchies

We allow you to model the project relationships in a multi-project build as they really are for your problem domain. Gradle follows your layout not vice versa.

Partial Builds

With Maven multi-project builds only work if executed from the root project and thus requiring a complete build. If you build from a subproject, only the subproject is built, not the projects the subproject depends on. Gradle offers partial builds. The subproject is built plus the projects it depends on. This is very convenient for larger builds.

Internal Groovy DSL

Gradle's build scripts are written in Groovy, not XML. This offers many advantages to XML: Rich interaction with existing libraries, ease of use, more power and a slower learning curve are some of them.

The Gradle Wrapper

The Gradle Wrapper allows you to execute Gradle builds on machines where Gradle is not installed. For example continuous integration servers or machines of users which want to build your open source project.

Gradle scales very well. It significantly increases your productivity, from rather simple single project builds up to huge enterprise multi-project builds.

Gradle is build by Gradle. From a build perspective Gradle is a simple project. But achieving the high degree of automation we have, would have been very hard (and expensive) to achieve with Ant or Maven.

2.2. Why Groovy?

We think the advantages of an internal DSL (based on a dynamic language) over XML are tremendous in case of *build scripts*. There are a couple of dynamic languages out there. Why Groovy? The answer lies in the context Gradle is operating in. Although Gradle is a general purpose build tool at its core, its main focus are Java projects. ^[2] In such projects obviously the team members know Java. One problem we see with Ant ^[3] and Maven is, that it involves a lot of knowledge only available to the build master. Such builds are very hard to comprehend, let alone to modify by a person not deeply involved with those tools. We think a build should be as transparent as possible to *all* team members.

You might argue why not using Java then as the language for build scripts. We think this is a valid question. It would have the highest transparency for your team and the lowest learning curve. But due to limitations of Java such a build language would not be as nice, expressive and powerful as it could be. ^[4] Languages like Python, Groovy or Ruby do a much better job here. We have chosen Groovy as it offers by far the highest transparency for Java people. Its base syntax is the same as Java's as well as its type system, its package structure other things. Groovy builds a lot on top of that. But on a common ground with Java.

For Java teams which share also Python or Ruby knowledge or are happy to learn it the above arguments don't apply. In the near future Gradle wants to give you a choice between different languages for your build scripts. For Jython or JRuby this should be easy to implement. If members of those communities are

interested in joining this effort, this is very much appreciated.

2.3. Missing features

Here a list of features you might expect but are not available yet:

- Creating IDE project and classpath files for IntelliJ and NetBeans. Gradle supports IDE project file generation for Eclipse.
- Integration with code coverage tools, such as Emma or Cobertura, and static analysis tools, such as Checkstyle, in our build-by-convention framework. Right now you have to integrate them yourself (for example using the Ant tasks for those tools).

[1] We mean Ant's targets here.

[2] Gradle also supports Groovy projects. Gradle will support Scala projects in a future release.

[3] If the advanced features are used (e.g. mixins, macrodefs, ...)

[4] At <http://www.defmacro.org/ramblings/lisp.html> you find an interesting article comparing Ant, XML, Java and Lisp. It's funny that the 'if Java had that syntax' syntax in this article is actually the Groovy syntax.

Getting Started

3.1. Prerequisites

Gradle requires a Java JDK to be installed. Gradle ships with its own Groovy library, therefore no Groovy needs to be installed. Any existing Groovy installation is ignored by Gradle. The standard Gradle distribution requires a JDK 1.5 or higher. We also provide a distinct JDK 1.4 compatible distribution.

Gradle uses whichever JDK it finds in your path (to check, use `java -version`). Alternatively, you can set the `JAVA_HOME` environment variable to point to the install directory of the desired JDK.

3.2. Download

You can download one of the Gradle distributions from the [Gradle web site](#).

3.3. Unpacking

The Gradle distribution comes packaged as a ZIP. The full distribution contains:

- The Gradle binaries.
- The user guide (HTML and PDF).
- The API documentation (Javadoc and Groovydoc).
- Extensive samples, including the examples referenced in the user guide, along with some complete and more complex builds you can use the starting point for your own build.
- The binary sources (If you want to build Gradle you need to download the source distribution or checkout the sources from the source repository).

For Un*x users

You need a GNU compatible tool to unzip Gradle, if you want the file permissions to be properly set. We mention this as some zip front ends for Mac OS X don't restore the file permissions properly.

3.4. Environment variables

For running Gradle, add `GRADLE_HOME/bin` to your `PATH` environment variable. Usually, this is sufficient to run Gradle. Optionally, you may also want to set the `GRADLE_HOME` environment variable to point to the root directory of your Gradle installation.

3.5. Running and testing your installation

You run Gradle via the **gradle** command. To check if Gradle is properly installed just type **gradle -v** and you should get an output like:

```
-----  
Gradle 0.8  
-----  
  
Gradle buildtime: Monday, September 28, 2009 1:53:43 PM CEST  
Groovy: 1.6.4  
Ant: Apache Ant version 1.7.0 compiled on December 13 2006  
Ivy: 2.1.0-rc2  
Java: 1.6.0_15  
JVM: 14.1-b02-90  
JVM Vendor: Apple Inc.  
OS Name: Mac OS X
```

3.6. JVM options

JVM options for running Gradle can be set via environment variables. You can use `GRADLE_OPTS` or `JAVA_OPTS`. Those variables can be used together. `JAVA_OPTS` is by convention an environment variable shared by many Java applications. A typical use case would be to set the HTTP proxy in `JAVA_OPTS` and the memory options in `GRADLE_OPTS`. Those variables can also be set at the beginning of the **gradle** or **gradlew** script.

4

Build Script Basics

You run a build using the **gradle** command. When run, **gradle** looks for a file called `build.gradle` in the current directory. [5] We call this `build.gradle` file a *build script*, although strictly speaking it is a build configuration script, as we will see later. In Gradle the build script defines a project. The name of the directory containing the build script is used as the name of the project.

4.1. Hello world

In Gradle the most basic building block is the *task*. The tasks for your build are defined in the build script. To try this out, create the following build script named `build.gradle`.

Example 4.1. The first build script

`build.gradle`

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

In a command-line shell, enter into the containing directory and execute the build script by running **gradle** -
:

Example 4.2. Execution of a build script

Output of **gradle -q hello**

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

What's going on here? This build file defines a single task, called `hello`, and adds an action to it. When you run **gradle hello**, Gradle executes the `hello` task, which in turn executes the action you've provided. The action is simply a closure containing some Groovy code to execute.

If you think this looks similar to Ant's targets, well, you are right. Gradle tasks are the equivalent to Ant targets. But as you will see, they are much more powerful. We have used a different

What does `-q` do?

Most of the examples in this user guide are run with the `-q` command-line option. This suppresses Gradle's log messages, so that only the output of the tasks is shown. You don't need to use this option if you don't want. See [Chapter 15, Logging](#) for more details about the command-line options which affect Gradle's output.

terminology than Ant as we think the word *task* is more expressive than the word *target*. Unfortunately this introduces a terminology clash with Ant, as Ant calls its commands, such as `javac` or `copy`, tasks. So when we talk about tasks, we *always* mean Gradle tasks, which are the equivalent to Ant's targets. If we talk about Ant tasks (Ant commands), we explicitly say *ant* task.

4.2. Build scripts are code

Gradle's build scripts expose to you the full power of Groovy. As an appetizer, have a look at this:

Example 4.3. Using Groovy in Gradle's tasks

build.gradle

```
task upper << {
    String someString = 'mY_nAmE'
    println "Original: " + someString
    println "Upper case: " + someString.toUpperCase()
}
```

Output of `gradle -q upper`

```
> gradle -q upper
Original: mY_nAmE
Upper case: MY_NAME
```

or

Example 4.4. Using Groovy in Gradle's tasks

build.gradle

```
task count << {
    4.times { print "$it " }
}
```

Output of `gradle -q count`

```
> gradle -q count
0 1 2 3
```

4.3. Task dependencies

As you probably have guessed, you can declare dependencies between your tasks.

Example 4.5. Declaration of dependencies between tasks

build.gradle

```
task hello << {
    println 'Hello world!'
}
task intro(dependsOn: hello) << {
    println "I'm Gradle"
}
```

Output of **gradle -q intro**

```
> gradle -q intro
Hello world!
I'm Gradle
```

To add a dependency, the corresponding task does not need to exist.

Example 4.6. Lazy dependsOn - the other task does not exist (yet)

build.gradle

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

Output of **gradle -q taskX**

```
> gradle -q taskX
taskY
taskX
```

The dependency of `taskX` to `taskY` is declared before `taskY` is defined. This is very important for multi-project builds. Task dependencies are discussed in more detail in [Section 13.4, “Adding dependencies to a task”](#).

Please notice, that you can't use a shortcut notation (see [Section 4.6, “Shortcut notations”](#)) when referring to task, which is not defined yet.

4.4. Dynamic tasks

The power of Groovy can be used for more than defining what a task does. For example, you can also use it to dynamically create tasks.

Example 4.7. Dynamic creation of a task

build.gradle

```
4.times { counter ->
    task "task$counter" << {
        println "I'm task number $counter"
    }
}
```

Output of **gradle -q task1**

```
> gradle -q task1
I'm task number 1
```

4.5. Manipulating existing tasks

Once tasks are created they can be accessed via an *API*. This is different to Ant. For example you can create additional dependencies.

Example 4.8. Accessing a task via API - adding a dependency

build.gradle

```
4.times { counter ->
    task "task$counter" << {
        println "I'm task number $counter"
    }
}
task0.dependsOn task2, task3
```

Output of **gradle -q task0**

```
> gradle -q task0
I'm task number 2
I'm task number 3
I'm task number 0
```

Or you can add behavior to an existing task.

Example 4.9. Accessing a task via API - adding behaviour

build.gradle

```
task hello << {
    println 'Hello Earth'
}
hello.doFirst {
    println 'Hello Venus'
}
hello.doLast {
    println 'Hello Mars'
}
hello << {
    println 'Hello Jupiter'
}
```

Output of **gradle -q hello**

```
> gradle -q hello
Hello Venus
Hello Earth
Hello Mars
Hello Jupiter
```

The calls `doFirst` and `doLast` can be executed multiple times. They add an action to the beginning or the end of the task's actions list. When the task executes, the actions in the action list are executed in order. The `<<` operator is simply an alias for `doLast`.

4.6. Shortcut notations

As you might have noticed in the previous examples, there is a convenient notation for accessing an *existing* task. Each task is available as a property of the build script:

Example 4.10. Accessing task as a property of the build script

build.gradle

```
task hello << {
    println 'Hello world!'
}
hello.doLast {
    println "Greetings from the $hello.name task."
}
```

Output of **gradle -q hello**

```
> gradle -q hello
Hello world!
Greetings from the hello task.
```

This enables very readable code, especially when using the out of the box tasks provided by the plugins (e.g. `compile`).

4.7. Dynamic task properties

You can assign arbitrary *new* properties to any task.

Example 4.11. Assigning properties to a task

build.gradle

```
task myTask
myTask.myProperty = 'myCustomPropValue'

task showProps << {
    println myTask.myProperty
}
```

Output of **gradle -q showProps**

```
> gradle -q showProps
myCustomPropValue
```

4.8. Using Ant Tasks

Ant tasks are first-class citizens in Gradle. Gradle provides excellent integration for Ant tasks simply by relying on Groovy. Groovy is shipped with the fantastic `AntBuilder`. Using Ant tasks from Gradle is as convenient and more powerful than using Ant tasks from a `build.xml` file. Let's look at an example:

Example 4.12. Using `AntBuilder` to execute `ant.checksum` target

build.gradle

```
task checksum << {
    def files = file('../antChecksumFiles').listFiles().sort()
    files.each { File file ->
        if (file.isFile()) {
            ant.checksum(file: file, property: file.name)
            println "$file.name Checksum: ${ant.properties[file.name]}"
        }
    }
}
```

Output of **gradle -q checksum**

```
> gradle -q checksum
agile_manifesto.html Checksum: 2dd24e01676046d8dedc2009a1a8f563
agile_principles.html Checksum: 659d204c8c7ccb5d633de0b0d26cd104
dylan_thomas.txt Checksum: 91040calcefcfbfdc8016b1b3e51f23d3
```

There is lots more you can do with Ant in your build scripts. You can find out more in [Chapter 16, Using Ant from Gradle](#).

4.9. Using methods

Gradle scales in how you can organize your build logic. The first level of organizing your build logic for the example above, is extracting a method.

Example 4.13. Using methods to organize your build logic

build.gradle

```
task checksum << {
    fileList('..antChecksumFiles').each {File file ->
        ant.checksum(file: file, property: "cs_${file.name}")
        println "${file.name} Checksum: ${ant.properties["cs_${file.name}"]}"
    }
}

task length << {
    fileList('..antChecksumFiles').each {File file ->
        ant.length(file: file, property: "lt_${file.name}")
        println "${file.name} Length: ${ant.properties["lt_${file.name}"]}"
    }
}

File[] fileList(String dir) {
    file(dir).listFiles({file -> file.isFile() } as FileFilter).sort()
}
```

Output of **gradle -q checksum**

```
> gradle -q checksum
agile_manifesto.html Checksum: 2dd24e01676046d8dedc2009a1a8f563
agile_principles.html Checksum: 659d204c8c7ccb5d633de0b0d26cd104
dylan_thomas.txt Checksum: 91040calcefcfbdc8016b1b3e51f23d3
```

Later you will see that such methods can be shared among subprojects in multi-project builds. If your build logic becomes more complex, Gradle offers you other very convenient ways to organize it. We have devoted a whole chapter to this. See [Chapter 34, *Organizing Build Logic*](#).

4.10. Default tasks

Gradle allows you to define one or more default tasks for your build.

Example 4.14. Defining a default tasks

build.gradle

```
defaultTasks 'clean', 'run'

task clean << {
    println 'Default Cleaning!'
}

task run << {
    println 'Default Running!'
}

task other << {
    println "I'm not a default task!"
}
```

Output of **gradle -q**

```
> gradle -q
Default Cleaning!
Default Running!
```

This is equivalent to running **gradle clean run**. In a multi-project build every subproject can have its own specific default tasks. If a subproject does not specify default tasks, the default tasks of the parent project are used (if defined).

4.11. Configure by DAG

As we describe in full detail later (See [Chapter 30, *The Build Lifecycle*](#)) Gradle has a configuration phase and an execution phase. After the configuration phase Gradle knows all tasks that should be executed. Gradle offers you a hook to make use of this information. A use-case for this would be to check if the release task is part of the tasks to be executed. Depending on this you can assign different values to some variables.

In the following example, execution of `distribution` and `release` tasks results in different value of `version` variable.

Example 4.15. Different outcomes of build depending on chosen tasks

build.gradle

```
gradle.taskGraph.whenReady {taskGraph ->
    if (taskGraph.hasTask(':release')) {
        version = '1.0'
    } else {
        version = '1.0-SNAPSHOT'
    }
}

task distribution << {
    println "We build the zip with version=$version"
}
task release(dependsOn: 'distribution') << {
    println 'We release now'
}
```

Output of `gradle -q distribution`

```
> gradle -q distribution
We build the zip with version=1.0-SNAPSHOT
```

Output of `gradle -q release`

```
> gradle -q release
We build the zip with version=1.0
We release now
```

The important thing is, that the fact that the release task has been chosen, has an effect *before* the release task gets executed. Nor has the release task to be the *primary* task (i.e. the task passed to the **gradle** command).

4.12. Summary

This is not the end of the story for tasks. So far we have worked with simple tasks. Tasks will be revisited in [Chapter 13, *More about Tasks*](#) and when we look at the Java Plugin in [Chapter 18, *The Java Plugin*](#).

[5] There are command line switches to change this behavior. See [Appendix B, *Gradle Command Line*](#)

Java Quickstart

As we have seen, Gradle is a general-purpose build tool. It can build pretty much anything you care to implement in your build script. Out-of-the-box, however, it doesn't build anything unless you write code in your build script. Most Java projects are pretty similar as far as the basics go: you need to compile your Java source files, run some unit tests, and create a JAR file containing your classes. It would be nice if you didn't have to code all this up for every project. Luckily, you don't have to. Gradle solves this problem through the use of *plugins*. A plugin is an extension to Gradle which configures your project in some way, typically by adding a bunch of pre-configured tasks which together do something useful. Gradle ships with a number of plugins, and you can easily write your own and share them with others. One such plugin is the *Java Plugin*. This plugin adds some tasks to your project which will compile and test your Java source code, and bundle it into a JAR file.

The Java plugin is convention based. This means that the plugin defines default values for many aspects of the project, such as where the Java source files are located. If you follow the convention in your project, you generally don't need to do much in your build script to get a useful build. Gradle allows you to customize your project if you don't want to or cannot follow the convention in some way. In fact, because support for Java projects is implemented as a plugin, you don't have to use the plugin at all to build a Java project, if you don't want to.

We have in-depth coverage with many examples about the Java plugin, dependency management and multi-project builds in later chapters. In this chapter we want to give you an initial idea of how to use the Java plugin to build a Java project.

5.1. A basic Java project

Let's look at a simple example. To use the Java plugin, add the following to your build file:

Example 5.1. Using the Java plugin

build.gradle

```
usePlugin 'java'
```

Note: The code for this example can be found at `samples/java/quickstart`

This is all you need to define a Java project. This will apply the Java plugin to your project, which adds a number of tasks to your project.

Gradle expects to find your production source code under `src/main/java` and your test source code under `src/test/java`. In addition, any files under `src/main/resources` will be included in the JAR file as resources, and any files under `src/test/resources` will be included in the classpath used to run the tests. All output files are created under the `build` directory, with the JAR file ending up in the `build/libs` directory.

What tasks are available?

You can use `gradle -t` to list the tasks of a project. This will let you see the tasks that the Java plugin has added to your project.

5.1.1. Building the project

The Java plugin adds quite a few tasks to your project. However, there are only a handful of tasks that you will need to use to build the project. The most commonly used task is the `build` task, which does a full build of the project. When you run `gradle build`, Gradle will compile and test your code, and create a JAR file containing your main classes and resources:

Example 5.2. Building a Java project

Output of `gradle build`

```
> gradle build
:compileJava
:processResources
:classes
:jar
:assemble
:compileTestJava
:processTestResources
:testClasses
:test
:check
:build

BUILD SUCCESSFUL

Total time: 1 secs
```

Some other useful tasks are:

clean

Deletes the `build` directory, removing all built files.

assemble

Compiles and jars your code, but does not run the unit tests. Other plugins add more artifacts to this task. For example, if you use the War plugin, this task will also build the WAR file for your project.

check

Compiles and tests your code. Other plugins add more checks to this task. For example, if you use the Code-quality plugin, this task will also run Checkstyle against your source code.

5.1.2. External dependencies

Usually, a Java project will have some dependencies on external JAR files. To reference these JAR files in the project, you need to tell Gradle where to find them. In Gradle, artifacts such as JAR files, are located in a *repository*. A repository can be used for fetching the dependencies of a project, or for publishing the artifacts of a project, or both. For this example, we will use the public Maven repository:

Example 5.3. Adding Maven repository

build.gradle

```
repositories {  
    mavenCentral()  
}
```

Let's add some dependencies. Here, we will declare that our production classes have a compile-time dependency on commons collections, and that our test classes have a compile-time dependency on junit:

Example 5.4. Adding dependencies

build.gradle

```
dependencies {  
    compile group: 'commons-collections', name: 'commons-collections', version: '3.2'  
    testCompile group: 'junit', name: 'junit', version: '4.+'  
}
```

You can find out more in [Chapter 28, Dependency Management](#).

5.1.3. Customising the project

The Java plugin adds a number of properties to your project. These properties have default values which are usually sufficient to get started. It's easy to change these values if they don't suit. Let's look at this for our sample. Here we will specify the version number for our Java project, along with the Java version our source is written in. We also add some attributes to the JAR manifest.

Example 5.5. Customization of MANIFEST.MF

build.gradle

```
sourceCompatibility = 1.5  
version = '1.0'  
manifest.mainAttributes(  
    'Implementation-Title': 'Gradle Quickstart',  
    'Implementation-Version': version  
)
```

The tasks which the Java plugin adds are regular tasks, exactly the same as if they were declared in the build file. This means you can use any of the mechanisms shown in earlier chapters to customise these tasks. For example, you can set the properties of a task, add behaviour to a task, change the dependencies of a task, or replace a task entirely. In our sample, we will configure the `test` task, which is of type `Test`, to add a system property when the tests are executed:

What properties are available?

You can use `gradle -r` to list the properties of a project. This will allow you to see the properties added by the Java plugin, and their default values.

Example 5.6. Adding system property

build.gradle

```
test {  
    options.systemProperties['property'] = 'value'  
}
```

5.1.4. Publishing the JAR file

Usually the JAR file needs to be published somewhere. To do this, you need to tell Gradle where to publish the JAR file. In Gradle, artifacts such as JAR files are published to repositories. In our sample, we will publish to a local directory. You can also publish to a remote location, or multiple locations.

Example 5.7. Publishing the JAR file

build.gradle

```
uploadArchives {  
    repositories {  
        flatDir(dirs: file('repos'))  
    }  
}
```

To publish the JAR file, run `gradle uploadArchives`.

5.1.5. Creating Eclipse project

To import your project into Eclipse, you need to add another plugin to your build file:

Example 5.8. Eclipse plugin

build.gradle

```
usePlugin 'eclipse'
```

Now execute `gradle eclipse` command to generate Eclipse project files. More on Eclipse task can be found in [Section 18.15, “Eclipse”](#).

5.1.6. Summary

Here's the complete build file for our sample:

Example 5.9. Java example - complete build file

build.gradle

```
usePlugin 'java'

usePlugin 'eclipse'

sourceCompatibility = 1.5
version = '1.0'
manifest.mainAttributes(
    'Implementation-Title': 'Gradle Quickstart',
    'Implementation-Version': version
)

repositories {
    mavenCentral()
}

dependencies {
    compile group: 'commons-collections', name: 'commons-collections', version: '3.2'
    testCompile group: 'junit', name: 'junit', version: '4.+'
}

test {
    options.systemProperties['property'] = 'value'
}

uploadArchives {
    repositories {
        flatDir(dirs: file('repos'))
    }
}
```

5.2. Multi-project Java build

Now let's look at a typical multi-project build. Below is the layout for the project:

Example 5.10. Multi-project build - hierarchical layout

Build layout

```
multiproject/
  api/
  services/
    webservice/
  shared/
```

Note: The code for this example can be found at `samples/java/multiproject`

Here we have three projects. Project `api` produces a JAR file which is shipped to the client to provide them a Java client for your XML webservice. Project `webservice` is a webapp which returns XML. Project `shared` contains code used both by `api` and `webservice`.

5.2.1. Defining a multi-project build

To define a multi-project build, you need to create a *settings file*. The settings file lives in the root directory of the source tree, and specifies which projects to include in the build. It must be called `settings.gradle`. For this example, we are using a simple hierarchical layout. Here is the corresponding settings file:

Example 5.11. Multi-project build - settings.gradle file

settings.gradle

```
include "shared", "api", "services:webservice"
```

You can find out more about the settings file in [Chapter 31, Multi-project Builds](#).

5.2.2. Common configuration

For most multi-project builds, there is some configuration which is common to all projects. In our sample, we will define this common configuration in the root project, using a technique called *configuration injection*. Here, the root project is like a container and the `subprojects` method iterates over the elements of this container - the projects in this instance - and injects the specified configuration. This way we can easily define the manifest content for all archives, and some common dependencies:

Example 5.12. Multi-project build - common configuration

build.gradle

```
subprojects {
    usePlugin 'java'
    usePlugin 'eclipse'

    repositories {
        mavenCentral()
    }

    dependencies {
        testCompile 'junit:junit:4.7'
    }

    group = 'org.gradle'
    version = '1.0'
    manifest.mainAttributes(provider: 'gradle')
}

dependsOnChildren()
```

Notice that our sample applies the Java plugin to each subproject. This means the tasks and configuration properties we have seen in the previous section are available in each subproject. So, you can compile, test, and JAR all the projects by running `gradle build` from the root project directory.

5.2.3. Dependencies between projects

You can add dependencies between projects in the same build, so that, for example, the JAR file of one project is used to compile another project. In the `api` build file we will add a dependency on the JAR produced by the `shared` project. Due to this dependency, Gradle will ensure that project `shared` always gets built before project `api`.

Example 5.13. Multi-project build - dependencies between projects

api/build.gradle

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

See [Section 31.7.1, “Disable the build of dependency projects.”](#) for how to disable this functionality.

5.2.4. Creating a distribution

We also add a distribution, that gets shipped to the client:

Example 5.14. Multi-project build - distribution file

api/build.gradle

```
task dist(type: Zip) {  
    dependsOn spiJar  
    from configurations.runtime  
    from spiJar.archivePath  
    fileSet dir: 'src/dist'  
}
```

5.3. Summary

In this chapter, you have seen how to do some of the things you commonly need to build a Java based project. This chapter is not exhaustive, and there are many other things you can do with Java projects in Gradle. These are dealt with in later chapters. Also, a lot of the behaviour you have seen in this chapter is configurable. For example, you can change where Gradle looks Java source files, or add extra tasks, or you can change what any task actually does. Again, you will see how this works in later chapters.

You can find out more about the Java plugin in [Chapter 18, *The Java Plugin*](#), and you can find more sample Java projects in the `samples/java` directory in the Gradle distribution.

6

Groovy Quickstart

To build a Groovy project, you use the *Groovy Plugin*. This plugin extends the Java plugin to add Groovy compilation capabilities to your project. Your project can contain Groovy source code, Java source code, or a mix of the two. In every other respect, a Groovy project is identical to a Java project, which we have already seen in Chapter 5, *Java Quickstart*.

6.1. A basic Groovy project

Let's look at an example. To use the Groovy plugin, add the following to your build file:

Example 6.1. Groovy plugin

build.gradle

```
usePlugin 'groovy'
```

Note: The code for this example can be found at `samples/groovy/quickstart`

This will also apply the Java plugin to the project, if it has not already been applied. The Groovy plugin extends the `compile` task to look for source files in directory `src/main/groovy`, and the `compileTest` task to look for test source files in directory `src/test/groovy`. The compile tasks use joint compilation for these directories, which means they can contain a mixture of java and groovy source files.

To use the groovy compilation tasks, you must also declare the Groovy version to use and where to find the Groovy libraries. You do this by adding a dependency to the `groovy` configuration. The `compile` configuration inherits this dependency, so the groovy libraries will be included in classpath when compiling Groovy and Java source. For our sample, we will use Groovy 1.6.0 from the public Maven repository:

Example 6.2. Dependency on Groovy 1.6.0

build.gradle

```
repositories {
    mavenCentral()
}

dependencies {
    groovy group: 'org.codehaus.groovy', name: 'groovy', version: '1.6.0'
}
```

Here is our complete build file:

Example 6.3. Groovy example - complete build file

build.gradle

```
usePlugin 'groovy'

repositories {
    mavenCentral()
}

dependencies {
    groovy group: 'org.codehaus.groovy', name: 'groovy', version: '1.6.0'
    testCompile group: 'junit', name: 'junit', version: '4.7'
}
```

Running `gradle build` will compile, test and JAR your project.

6.2. Summary

This chapter describes a very simple Groovy project. Usually, a real project will require more than this. Because a Groovy project *is* a Java project, whatever you can do with a Java project, you can also do with a Groovy project.

You can find out more about the Groovy plugin in [Chapter 19, *The Groovy Plugin*](#), and you can find more sample Groovy projects in the `samples/groovy` directory in the Gradle distribution.

Web Application Quickstart

This chapter is a work in progress.

This chapter introduces some of the Gradle's support for web applications. Gradle provides two plugins for web application development: the War plugin and the Jetty plugin. The War plugin extends the Java plugin to build a WAR file for your project. The Jetty plugin extends the War plugin to allow you to deploy your web application to an embedded Jetty web container.

7.1. Building a WAR file

To build a WAR file, you apply the War plugin to your project:

Example 7.1. War plugin

build.gradle

```
usePlugin 'war'
```

Note: The code for this example can be found at `samples/webApplication/quickstart`

This also applies the Java plugin to your project. Running `gradle build` will compile, test and WAR your project. Gradle will look for the source files to include in the WAR file in `src/main/webapp`. Your compiled classes, and their runtime dependencies are also included in the WAR file.

7.2. Running your web application

To run your web application, you apply the Jetty plugin to your project:

Example 7.2. Running web application with Jetty plugin

build.gradle

```
usePlugin 'jetty'
```

Groovy web applications

You can combine multiple plugins in a single project, so you can use the War and Groovy plugins together to build a Groovy based web application. The appropriate groovy libraries will be added to the WAR file for you.

This also applies the War plugin to your project. Running `gradle jettyRun`

will run your web application in an embedded Jetty web container. Running `gradle jettyRunWar` will build and test the WAR file, and then run it in an embedded web container.

TODO: which url, configure port, uses source files in place and can edit your files and reload.

7.3. Summary

You can find out more about the War plugin in [Chapter 21, *The War Plugin*](#) and the Jetty plugin in [Chapter 22, *The Jetty Plugin*](#). You can find more sample Java projects in the `samples/webApplication` directory in the Gradle distribution.

Artifact Basics

This chapter is currently under construction.

For all the details of artifact handling see [Chapter 29, *Artifact Management*](#).

This chapter introduces some of the basics of artifact handling in Gradle.

8.1. Artifact configurations

Artifacts are grouped into *configurations*. A configuration is simply a set of files with a name. You can use them to declare the external dependencies your project has, or to declare the artifacts which your project publishes.

To define a configuration:

Example 8.1. Definition of a configuration

build.gradle

```
configurations {  
    compile  
}
```

To access a configuration:

Example 8.2. Accessing a configuration

build.gradle

```
println configurations.compile.name  
println configurations['compile'].name
```

To configure a configuration:

Example 8.3. Configuration of a configuration

build.gradle

```
configurations {
    compile {
        description = 'compile classpath'
        transitive = true
    }
    runtime {
        extendsFrom compile
    }
}
configurations.compile {
    description = 'compile classpath'
}
```

8.2. Repositories

Artifacts are stored in *repositories*.

To use maven central repository:

Example 8.4. Usage of Maven central repository

build.gradle

```
repositories {
    mavenCentral()
}
```

To use a local directory:

Example 8.5. Usage of a local directory

build.gradle

```
repositories {
    flatDir name: 'localRepository', dirs: 'lib'
}
```

You can also use any Ivy resolver. You can have multiple repositories.

To access a repository:

Example 8.6. Accessing a repository

build.gradle

```
println repositories.localRepository.name
println repositories['localRepository'].name
```

To configure a repository:

Example 8.7. Configuration of a repository

build.gradle

```
repositories {
    localRepository {
        addArtifactPattern(file('lib').absolutePath + '/[name]/[revision]/[name]-[revision]')
    }
}
repositories.localRepository {
    addArtifactPattern(file('lib').absolutePath + '/[name]/[revision]/[name]-[revision]')
```

8.3. External dependencies

To define an external dependency, you add a dependency to a configuration:

Example 8.8. Definition of an external dependency

build.gradle

```
configurations {
    compile
}

dependencies {
    compile group: 'commons-collections', name: 'commons-collections', version: '3.2'
```

group and version are optional

TBD - configuring an external dependency

To use the external dependencies of a configuration:

Example 8.9. Usage of external dependency of a configuration

build.gradle

```
task listJars << {
    configurations.compile.each { File file -> println file.name }
}
```

Output of **gradle -q listJars**

```
> gradle -q listJars
commons-collections-3.2.jar
```

8.4. Artifact publishing

TBD

8.5. API

Configurations are contained in a ConfigurationContainer . Each configuration implements the Configuration.

Using the Gradle Command-Line

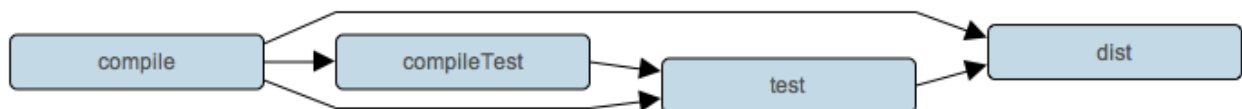
This chapter introduces the basics of the Gradle command-line. You run a build using the **gradle** command, which you have already seen in action in previous chapters.

9.1. Executing multiple tasks

You can execute multiple tasks in a single build by listing each of the tasks on the command-line. For example, the command **gradle compile test** will execute the `compile` and `test` tasks. Gradle will execute the tasks in the order that they are listed on the command-line, and will also execute the dependencies for each task. Each task is executed once only, regardless of why it is included in the build: whether it was specified on the command-line, or it a dependency of another task, or both. Let's look at an example.

Below four tasks are defined. Both `dist` and `test` depend on the `compile` task. Running **gradle dist t** for this build script results in the `compile` task being executed only once.

Figure 9.1. Task dependencies



Example 9.1. Executing multiple tasks

build.gradle

```
task compile << {
    println 'compiling source'
}

task compileTest(dependsOn: compile) << {
    println 'compiling unit tests'
}

task test(dependsOn: [compile, compileTest]) << {
    println 'running unit tests'
}

task dist(dependsOn: [compile, test]) << {
    println 'building the distribution'
}
```

Output of **gradle dist test**

```
> gradle dist test
:compile
compiling source
:compileTest
compiling unit tests
:test
running unit tests
:dist
building the distribution

BUILD SUCCESSFUL

Total time: 1 secs
```

Because each task is executed once only, executing **gradle test test** is exactly the same as executing **gradle test**.

9.2. Excluding tasks

You can exclude a task from being executed using the **-x** command-line option and providing the name of the task to exclude. Let's try this with the sample build file above.

Example 9.2. Excluding tasks

Output of **gradle dist -x test**

```
> gradle dist -x test
:compile
compiling source
:dist
building the distribution

BUILD SUCCESSFUL

Total time: 1 secs
```

You can see from the output of this example, that the **test** task is not executed, even though it is a

dependency of the `dist` task. You will also notice that 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.

9.3. Task name abbreviation

When you specify tasks on the command-line, you don't have to provide the full name of the task. You only need to provide enough of the task name to uniquely identify the task. For example, in the sample build above, you can execute task `dist` by running **gradle d**:

Example 9.3. Abbreviated task name

Output of **gradle d**

```
> gradle d
:compile
compiling source
:compileTest
compiling unit tests
:test
running unit tests
:dist
building the distribution

BUILD SUCCESSFUL

Total time: 1 secs
```

You can also abbreviate each word in a camel case task name. For example, you can execute task `compile` by running **gradle compTest** or even **gradle cT**

Example 9.4. Abbreviated camel case task name

Output of **gradle cT**

```
> gradle cT
:compile
compiling source
:compileTest
compiling unit tests

BUILD SUCCESSFUL

Total time: 1 secs
```

You can also use these abbreviations with the `-x` command-line option.

9.4. Selecting which build to execute

When you run the **gradle** command, it looks for a build file in the current directory. You can use the `-b` option to select another build file. For example:

Example 9.5. Selecting the project using a build file

subdir/myproject.gradle

```
task hello << {
    println "using build file '$buildFile.name' in '$buildFile.parentFile.name'."
}
```

Output of **gradle -q -b subdir/myproject.gradle hello**

```
> gradle -q -b subdir/myproject.gradle hello
using build file 'myproject.gradle' in 'subdir'.
```

Alternatively, you can use the `-p` option to specify the project directory to use:

Example 9.6. Selecting the project using project directory

Output of **gradle -q -p subdir hello**

```
> gradle -q -p subdir hello
using build file 'build.gradle' in 'subdir'.
```

9.5. Obtaining information about your build

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

Running **gradle --tasks** gives you a list of the tasks which make up the build, broken down by project. This report shows the default tasks, if any, of each project, and the description and dependencies of each task. Below is an example of this report:

Example 9.7. Obtaining information about tasks

Output of **gradle -q --tasks**

```
> gradle -q --tasks
-----
Root Project
-----
Default Tasks: dists

:clean - Deletes the build directory (build)
:dists
  -> :api:libs, :webapp:libs

-----
Project :api
-----
:api:libs
rule - build<ConfigurationName>: builds the artifacts of the given configuration

-----
Project :webapp
-----
:webapp:libs
rule - build<ConfigurationName>: builds the artifacts of the given configuration
```

Running **gradle --dependencies** gives you a list of the dependencies of the build, broken down by project. This report shows the configurations of each project. For each configuration, the direct and transitive dependencies of that configuration are shown. Below is an example of this report:

Example 9.8. Obtaining information about dependencies

Output of **gradle -q --dependencies**

```
> gradle -q --dependencies
-----
Root Project
-----
No configurations

-----
Project :api
-----
compile
|-----junit:junit:4.7:default

-----
Project :webapp
-----
compile
|-----commons-io:commons-io:1.2:default
```

Running **gradle --properties** gives you a list of the properties of each project in the build.

You can also use the [project report plugin](#) to add a number of reporting tasks to your project.

9.6. Dry Run

Sometimes you are interested in which tasks are executed in which order for a given set of tasks specified on the command line, but you don't want the tasks to be executed. You can use the **-m** option for this. For example **gradle -m clean compile** shows you all tasks to be executed as part of the **clean** and **compile** tasks. This is complementary to the **-t**, which shows you all available tasks for execution.

9.7. Summary

In this chapter, you have seen some of the things you can do with Gradle from the command-line. You can find out more about the **gradle** command in [Appendix B, *Gradle Command Line*](#).

Using the Gradle Graphical User Interface

In addition to supporting a traditional command line interface, gradle offers a graphical user interface. This is a stand alone user interface that can be launched with the **--gui** option.

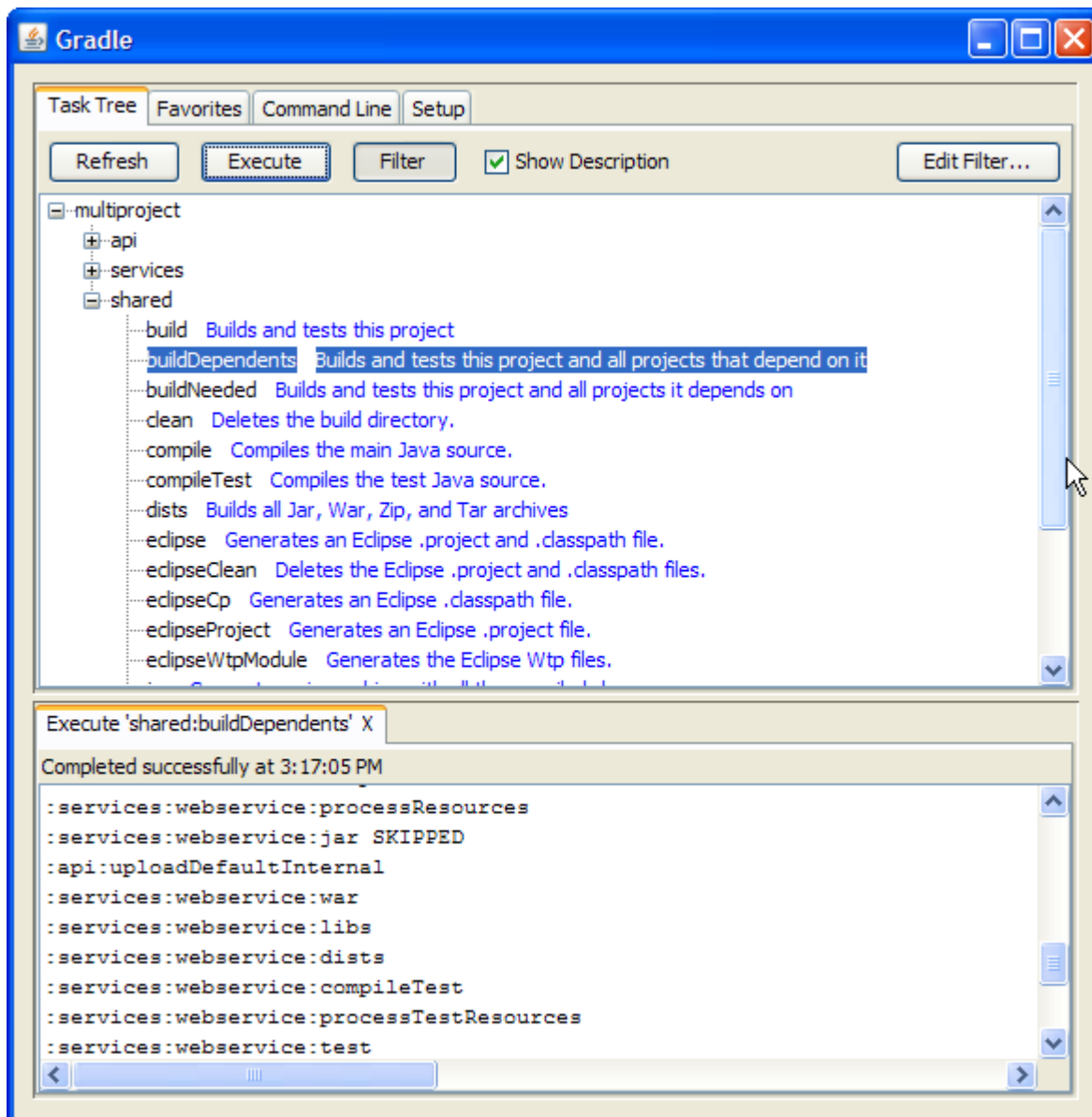
Example 10.1. Launching the GUI

```
gradle --gui
```

Note that this command blocks until the gradle GUI is closed. Under *nix it is probably preferable to run this as a background task (**gradle --gui&**)

If you run this from your gradle project working directory, you should see a tree of tasks.

Figure 10.1. GUI Task Tree



It is preferable to run this command from your gradle project directory so that the settings of the UI will be stored in your project directory. However, you can run it then change the working directory via the Setup tab in the UI.

The UI displays 4 tabs along the top and an output window along the bottom.

10.1. Task Tree

The Task Tree shows a hierarchical display of all projects and their tasks. Double clicking a task executes it.

There is also a filter so that uncommon tasks can be hidden. You can toggle the filter via the Filter button. Editing the filter allows you to configure which tasks and projects are shown. Hidden tasks show up in red. Note: newly created tasks will show up by default (versus being hidden by default).

The Task Tree context menu provides the following options:

- Execute ignoring dependencies. This does not require dependent projects to be rebuilt (same as the

-a option).

- Add tasks to the favorites (see Favorites tab)
- Hide the selected tasks. This adds them to the filter.
- Edit the build.gradle file. Note: this requires Java 1.6 or higher and requires that you have .gradle files associated in your OS.

10.2. Favorites

The Favorites tab is place to store commonly-executed commands. These can be complex commands (anything that's legal to gradle) and you can provide them with a display name. This is useful for creating, say, a custom build command that explicitly skips tests, documentation, and samples that you could call "fast build".

You can reorder favorites to your liking and even export them to disk so they can imported by others. If you edit them, you are given options to "Always Show Live Output." This only applies if you have 'Only Show Output When Errors Occur'. This override always forces the output to be shown.

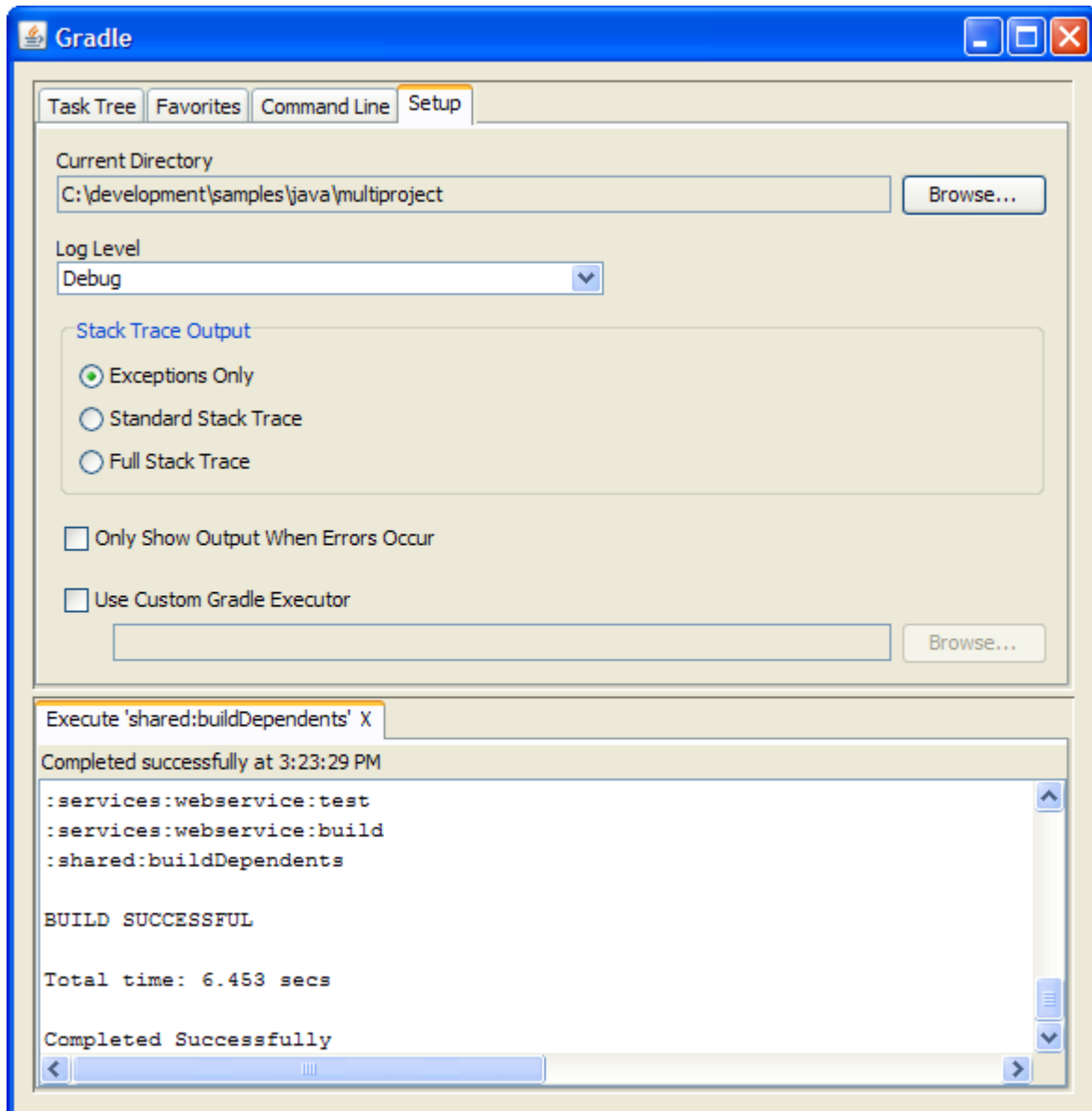
10.3. Command Line

The Command Line tab is place to execute a single gradle command directly. Just enter whatever you would normally enter after 'gradle' on the command line. This also provides a place to try out commands before adding them to favorites.

10.4. Setup

The Setup tab allows configuration of some general settings.

Figure 10.2. GUI Setup



- Current Directory

Defines the root directory of your gradle project (typically where build.gradle is located).

- Stack Trace Output

This determines how much information to write out stack traces when errors occur. Note: if you specify a stack trace level on either the Command Line or Favorites tab, it will override this stack trace level.

- Only Show Output When Errors Occur

Enabling this option hides any output when a task is executed unless the build fails.

- Use Custom Gradle Executor - Advanced feature

This provides you with an alternate way to launch gradle commands. This is useful if your project requires some extra setup that is done inside another batch file or shell script (such as specifying an

init script).

11

Tutorial - 'This and That'

11.1. Directory creation

There is a common situation, that multiple tasks depend on the existence of a directory. Of course you can deal with this by adding a `mkdir` to the beginning of those tasks. But this is kind of bloated. There is a better solution (works only if the tasks that need the directory have a *dependsOn* relationship):

Example 11.1. Directory creation with `mkdir`

build.gradle

```
classesDir = new File('build/classes')
task resources << {
    classesDir.mkdirs()
    // do something
}
task compile(dependsOn: 'resources') << {
    if (classesDir.isDirectory()) {
        println 'The class directory exists. I can operate'
    }
    // do something
}
```

Output of `gradle -q compile`

```
> gradle -q compile
The class directory exists. I can operate
```

But Gradle offers you also *Directory Tasks* to deal with this.

Example 11.2. Directory creation with Directory tasks

build.gradle

```
classes = dir('build/classes')
task resources(dependsOn: classes) << {
    // do something
}
task otherResources(dependsOn: classes) << {
    if (classes.dir.isDirectory()) {
        println 'The class directory exists. I can operate'
    }
    // do something
}
```

Output of `gradle -q otherResources`

```
> gradle -q otherResources
The class directory exists. I can operate
```

A *Directory Task* is a simple task whose name is a relative path to the project dir ^[6]. During the execution phase the directory corresponding to this path gets created if it does not exist yet. Another interesting thing to note in this example, is that you can also pass tasks objects to the `dependsOn` declaration of a task.

11.2. Gradle properties and system properties

Gradle offers a variety of ways to add properties to your build. With the `-D` command line option you can pass a system property to the JVM which runs Gradle. The `-D` option of the **gradle** command has the same effect as the `-D` option of the **java** command.

You can also directly add properties to your project objects using properties files. You can place a `gradle.p` file in the Gradle user home directory (defaults to `USER_HOME/.gradle`) or in your project directory. For multi-project builds you can place `gradle.properties` files in any subproject directory. The properties of the `gradle.properties` can be accessed via the project object. The properties file in the user's home directory has precedence over property files in the project directories.

You can also add properties directly to your project object via the `-P` command line option. For more exotic use cases you can even pass properties *directly* to the project object via system and environment properties. For example if you run a build on a continuous integration server where you have no admin rights for the *machine*. Your build script needs properties which values should not be seen by others. Therefore you can't use the `-P` option. In this case you can add an environment property in the project administration section (invisible to normal users). ^[7] If the environment property follows the pattern `ORG_GRADLE_PROJECT_prope`, `propertyName` is added to your project object. If in the future CI servers support Gradle directly, they might start Gradle via its main method. Therefore we already support the same mechanism for system properties. The only difference is the pattern, which is `org.gradle.project.propertyName`.

With the `gradle.properties` files you can also set system properties. If a property in such a file has the prefix `systemProp.` the property and its value are added to the system properties, without the prefix.

Example 11.3. Setting properties with a gradle.properties file

gradle.properties

```
gradlePropertiesProp=gradlePropertiesValue
systemPropertiesProp=shouldBeOverWrittenBySystemProp
envPropertiesProp=shouldBeOverWrittenByEnvProp
systemProp.system=systemValue
```

build.gradle

```
task printProps << {
    println commandLineProjectProp
    println gradlePropertiesProp
    println systemProjectProp
    println envProjectProp
    println System.properties['system']
}
```

Output of `gradle -q -PcommandLineProjectProp=commandLineProjectPropValue -Dorg.gradle.project.s`

```
> gradle -q -PcommandLineProjectProp=commandLineProjectPropValue -Dorg.gradle.project.s
commandLineProjectPropValue
gradlePropertiesValue
systemPropertyValue
envPropertyValue
systemValue
```

11.2.1. Checking for project properties

You can access a project property in your build script simply by using its name as you would use a variable. In case this property does not exist, an exception is thrown and the build fails. If your build script relies on optional properties the user might set for example in a gradle.properties file, you need to check for existence before you can access them. You can do this by using the method `hasProperty('propertyName')` which returns `true` or `false`.

11.3. Accessing the web via a proxy

Setting a proxy for web access (for example for downloading dependencies) is easy. Gradle does not need to provide special functionality for this. The JVM can be instructed to go via proxy by setting certain system properties. You could set these system properties directly in your build script with `System.properties['propertyName']`. An arguably nicer way is shown in [Section 11.2, “Gradle properties and system properties”](#). Your gradle.properties file could look like this:

Example 11.4. Accessing the web via a proxy

gradle.properties

```
systemProp.http.proxyHost=www.somehost.org
systemProp.http.proxyPort=8080
systemProp.http.proxyUser=userid
systemProp.http.proxyPassword=password
systemProp.http.nonProxyHosts=*.nonproxyrepos.com|localhost
```

We could not find a good overview for all possible proxy settings. One place to look are the constants in a file from the ant project. Here a [link to the svn view](#). The other is a [Networking Properties page](#) from the JDK

docs. If anyone knows a better overview please let us know via the mailing list.

11.4. Caching

To improve the responsiveness Gradle caches the compiled build script by default. The first time you run a build for a project, Gradle creates a `.gradle` directory in which it puts the compiled build script. The next time you run this build, Gradle uses the compiled build script, if the timestamp of the compiled script is newer than the timestamp of the actual build script. Otherwise the build script gets compiled and the new version is stored in the cache. If you run Gradle with the `-x` option, any existing cache is ignored and the build script is compiled and executed on the fly. If you run Gradle with the `-r` option, the build script is always compiled and stored in the cache. That way you can always rebuild the cache if for example the timestamps for some reasons don't reflect that the build script needs to be recompiled.

11.5. Configuring arbitrary objects

You can configure arbitrary objects in the following very readable way.

Example 11.5. Configuring arbitrary objects

build.gradle

```
task configure << {
    pos = configure(new java.text.FieldPosition(10)) {
        beginIndex = 1
        endIndex = 5
    }
    println pos.beginIndex
    println pos.endIndex
}
```

Output of `gradle -q configure`

```
> gradle -q configure
1
5
```

[6] The notation `dir('/somepath')` is a convenience method for `tasks.add('somepath', type: Dir)`

[7] *Teamcity* or *Bamboo* are for example CI servers which offer this functionality.

12

The Project and Task API

12.1. Project API

In the tutorial in [Chapter 4, *Build Script Basics*](#) we used, for example, the `task()` method. Where does this method come from? We said earlier that the build script defines a project in Gradle. For Gradle, this means that it creates an instance of `Project` and associates this `Project` object with the build script. As the build script executes, it configures this `Project` object.

- Any method you call in your build script, which *is not defined* in the build script, is delegated to the `Project` object.
- Any property you access in your build script, which *is not defined* in the build script, is delegated to the `Project` object.

Let's try this out and try to access the `name` property of the `Project` object.

Example 12.1. Accessing property of the `Project` object

`build.gradle`

```
task check << {  
    println name  
    println project.name  
}
```

Output of `gradle -q check`

```
> gradle -q check  
projectApi  
projectApi
```

Both `println` statements print out the same property. The first uses auto-delegation to the `Project` object, for properties not defined in the build script. The other statement uses the `project` property available to any build script, which returns the associated `Project` object. Only if you define a property or a method which has the same name as a member of the `Project` object, you need to use the `project` property.

Have a look at the [Project API](#) to find out more about project properties and methods.

12.1.1. Standard project properties

The `Project` object provides some standard properties, which are available in your build script. The following table lists a few of the commonly used ones.

Table 12.1. Project Properties

Name	Type	Default Value
<code>project</code>	<code>Project</code>	The <code>Project</code> instance
<code>name</code>	<code>String</code>	The name of the directory containing the build script.
<code>path</code>	<code>String</code>	The absolute path of the project.
<code>buildFile</code>	<code>File</code>	The build script.
<code>projectDir</code>	<code>File</code>	The directory containing the build script.
<code>buildDirName</code>	<code>String</code>	<code>build</code>
<code>buildDir</code>	<code>File</code>	<code>projectDir/build</code>
<code>group</code>	<code>Object</code>	unspecified
<code>version</code>	<code>Object</code>	unspecified
<code>ant</code>	<code>AntBuilder</code>	An <code>AntBuilder</code> instance

Below is a sample build which demonstrates some of these properties.

Example 12.2. Project properties

Build layout

```
projectCoreProperties/  
  build.gradle  
subProject/  
  build.gradle
```

build.gradle

```
task check << {  
    allprojects {  
        println "project path $path"  
        println "  project name = $name"  
        println "  project dir = '${rootProject.relativePath(projectDir)}'  
        println "  build file = '${rootProject.relativePath(buildFile)}'  
        println "  build dir = '${rootProject.relativePath(buildDir)}'  
    }  
}
```

Output of `gradle -q check`

```
> gradle -q check  
project path :  
  project name = projectCoreProperties  
  project dir = ''  
  build file = 'build.gradle'  
  build dir = 'build'  
project path :subProject  
  project name = subProject  
  project dir = 'subProject'  
  build file = 'subProject/build.gradle'  
  build dir = 'subProject/build'
```

12.2. Task API

Many of the methods of the `Project` instance return task objects. We have already seen some ways that you can use task objects in [Chapter 4, *Build Script Basics*](#). Look here to learn more about [Task](#).

12.3. Summary

The project and the task API constitute the core layer of Gradle and provide all the possible interaction options with this layer. ^[8] This core-layer constitutes a language for dependency based programming. ^[9] There are many other projects providing such a language. There is Ant for Java, Rake and Rant for Ruby, SCons for Python, the good old Make and many more. ^[10] We think that one thing that makes Gradle special compared to the other tools, is its strong support for applying dependency based programming on *multi-project* builds. We also think that just Gradle's core layer (together with its integration of the Ant tasks), provides a more convenient build system than Ant's core layer.

[8] There is more to come for this layer in the other chapters, e.g. support for multi-project builds (see [Chapter 31, *Multi-project Builds*](#)).

[9] Martin Fowler has written about this:
<http://martinfowler.com/articles/rake.html#DependencyBasedProgramming>

[10] Interestingly, Maven2 is the only major build system which does not use dependency based programming.

13

More about Tasks

In the introductory tutorial (Chapter 4, *Build Script Basics*) you have learned how to create simple tasks. You have also learned how to add additional behavior to these tasks later on. And you have learned how to create dependencies between tasks. This was all about simple tasks. But Gradle takes the concept of tasks further. Gradle supports *enhanced tasks*, that is, tasks which have their own properties and methods. This is really different to what you are used to with Ant targets. Such enhanced tasks are either provided by you or are provided by Gradle.

13.1. Defining tasks

We have already seen how to define tasks using a keyword style in Chapter 4, *Build Script Basics*. There are a few variations on this style, which you may need to use in certain situations. For example, the keyword style does not work in expressions.

Example 13.1. Defining tasks

build.gradle

```
task(hello) << {
    println "hello"
}

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

You can also use strings for the task names:

Example 13.2. Defining tasks - using strings

build.gradle

```
task('hello') <<
{
    println "hello"
}

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

There is an alternative syntax for defining tasks, which you may prefer to use:

Example 13.3. Defining tasks with alternative syntax

build.gradle

```
tasks.add(name: 'hello') << {
    println "hello"
}

tasks.add(name: 'copy', type: Copy) {
    from(file('srcDir'))
    into(buildDir)
}
```

Here we add tasks to the `tasks` collection. Have a look at [TaskContainer](#) for more variations of the `add()` method.

13.2. Locating tasks

You often need to locate the tasks that you have defined in the build file, for example, to configure them or use them for dependencies. There are a number of ways you can do this. Firstly, each task is available as a property of the project, using the task name as the property name:

Example 13.4. Accessing tasks as properties

build.gradle

```
task hello

println hello.name
println project.hello.name
```

Tasks are also available through the `tasks` collection.

Example 13.5. Accessing tasks via tasks collection

build.gradle

```
task hello

println tasks.hello.name
println tasks['hello'].name
```

You can access tasks from any project using the task's path using the `tasks.getByPath()` method. You can call the `getByPath()` method with a task name, or a relative path, or an absolute path.

Example 13.6. Accessing tasks by path

build.gradle

```
project(':projectA') {
    task hello
}

task hello

println tasks.getByPath('hello').path
println tasks.getByPath(':hello').path
println tasks.getByPath('projectA:hello').path
println tasks.getByPath(':projectA:hello').path
```

Output of `gradle -q hello`

```
> gradle -q hello
:hello
:hello
:projectA:hello
:projectA:hello
```

Have a look at [TaskContainer](#) for more options for locating tasks.

13.3. Configuring tasks

As an example, let's look at the `Copy` task provided by Gradle. To create a `Copy` task for your build, you can declare in your build script: [\[11\]](#)

Example 13.7. Creating a copy task

build.gradle

```
task myCopy(type: Copy)
```

This creates a copy task with no default behavior. The task can be configured using its API (see [Copy](#)). The following examples show several different ways to achieve the same configuration.

Example 13.8. Configuring a task - various ways

build.gradle

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

This is similar to the way we would normally configure objects in Java. You have to repeat the context (`myCopy`) in the configuration statement every time. This is a redundancy and not very nice to read.

There is a more convenient way of doing this.

Example 13.9. Configuring a task - fluent interface

build.gradle

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

You might know this approach from the Hibernate Criteria Query API or JMock. Of course the API of a task has to support this. The `from`, `to` and `include` methods all return an object that may be used to chain to additional configuration methods. Gradle's build-in tasks usually support this configuration style.

But there is yet another way of configuring a task. It also preserves the context and it is arguably the most readable. It is usually our favorite.

Example 13.10. Configuring a task - with closure

build.gradle

```
task myCopy(type: Copy)

myCopy {
    from 'resources'
    into 'target'
    include('**/*.txt', '**/*.xml', '**/*.properties')
}
```

This works for *any* task. Line 3 of the example is just a shortcut for the `tasks.getByName()` method. It is important to note that if you pass a closure to the `getByName()` method, this closure is applied to *configure* the task.

There is a slightly different ways of doing this.

Example 13.11. Configuring a task - with `configure()` method

build.gradle

```
task myCopy(type: Copy)

myCopy.configure {
    from('source')
    into('target')
    include('**/*.txt', '**/*.xml', '**/*.properties')
}
```

Every task has a `configure()` method, which you can pass a closure for configuring the task. Gradle uses this style for configuring objects in many places, not just for tasks.

You can also use a configuration closure when you define a task.

Example 13.12. Defining a task with closure

build.gradle

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

13.4. Adding dependencies to a task

There are several ways you can define the dependencies of a task. In [Section 4.3, “Task dependencies”](#) you were introduced to defining dependencies using task names. Task names can refer to tasks in the same project as the task, or to tasks in other projects. To refer to a task in another project, you prefix the name of the task with the path of the project it belongs to. Below is an example which adds a dependency from `projectB:taskY` to `projectA:taskX`:

Example 13.13. Adding dependency on task from another project

build.gradle

```
project('projectA') {
    task taskX(dependsOn: ':projectB:taskY') << {
        println 'taskX'
    }
}

project('projectB') {
    task taskY << {
        println 'taskY'
    }
}
```

Output of `gradle -q taskX`

```
> gradle -q taskX
taskY
taskX
```

Instead of using a task name, you can define a dependency using a `Task` object, as shown in this example:

Example 13.14. Adding dependency using task object

build.gradle

```
task taskX << {
    println 'taskX'
}

task taskY << {
    println 'taskY'
}

taskX.dependsOn taskY
```

Output of `gradle -q taskX`

```
> gradle -q taskX
taskY
taskX
```

For more advanced uses, you can define a task dependency using a closure. When evaluated, the closure is passed the task whose dependencies are being calculated. The closure should return a single `Task` or collection of `Task` objects, which are then treated as dependencies of the task. The following example adds a dependency from `taskX` to all the tasks in the project whose name starts with `lib`:

Example 13.15. Adding dependency using closure

build.gradle

```
task taskX << {
    println 'taskX'
}

taskX.dependsOn {
    tasks.findAll { task -> task.name.startsWith('lib') }
}

task lib1 << {
    println 'lib1'
}

task lib2 << {
    println 'lib2'
}

task notALib << {
    println 'notALib'
}
```

Output of `gradle -q taskX`

```
> gradle -q taskX
lib1
lib2
taskX
```

For more information about task dependencies, see the [Task API](#).

13.5. Adding a description to a task

You can add a description to your task. This description is for example displayed when executing `gradle -t`.

Example 13.16. Adding a description to a task

build.gradle

```
task copy(type: Copy) {
    description = 'Copies the resource directory to the target directory.'
    from 'resources'
    into 'target'
    include('**/*.txt', '**/*.xml', '**/*.properties')
}
```

13.6. Replacing tasks

Sometimes you want to replace a task. For example if you want to exchange a task added by the Java Plugin with a custom task of a different type. You can achieve this with:

Example 13.17. Overwriting a task

build.gradle

```
task copy(type: Copy)

task copy(overwrite: true) << {
    println('I am the new one.')
}
```

Output of `gradle -q copy`

```
> gradle -q copy
I am the new one.
```

Here we replace a task of type `Copy` with a simple task. When creating the simple task, you have to set the `overwrite` property to `true`. Otherwise Gradle throws an exception, saying that a task with such a name already exists.

13.7. Skipping tasks

Gradle offers multiple ways to skip the execution of a task.

13.7.1. Using a predicate

You can use the `onlyIf()` method to attach a predicate to a task. The task's actions are only executed if the predicate evaluates to `true`. You implement the predicate as a closure. The closure is passed the task as a parameter, and should return `true` if the task should execute and `false` if the task should be skipped. The predicate is evaluated just before the task is due to be executed.

Example 13.18. Skipping a task using a predicate

build.gradle

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

hello.onlyIf { !project.hasProperty('skipHello') }
```

Output of **gradle hello -PskipHello**

```
> gradle hello -PskipHello
:hello SKIPPED as onlyIf is false

BUILD SUCCESSFUL

Total time: 1 secs
```

13.7.2. Using StopExecutionException

If the rules for skipping a task can't be expressed with predicate, you can use the [StopExecutionException](#). If this exception is thrown by an action, the further execution of this action as well as the execution of any following action of this task is skipped. The build continues with executing the next task.

Example 13.19. Skipping tasks with StopExecutionException

build.gradle

```
task compile << {
    println 'We are doing the compile.'
}

compile.doFirst {
    // Here you would put arbitrary conditions in real life. But we use this as an inte
    if (true) { throw new StopExecutionException() }
}

task myTask(dependsOn: 'compile') << {
    println 'I am not affected'
}
```

Output of **gradle -q myTask**

```
> gradle -q myTask
I am not affected
```

This feature is helpful if you work with tasks provided by Gradle. It allows you to add *conditional* execution of the built-in actions of such a task. ^[12]

13.7.3. Enabling and disabling tasks

Every task has also an `enabled` flag which defaults to `true`. Setting it to `false` prevents the execution of any of the task's actions.

Example 13.20. Enabling and disabling tasks

build.gradle

```
task disableMe << {  
    println 'This should not be printed if the task is disabled.'  
}  
disableMe.enabled = false
```

Output of `gradle disableMe`

```
> gradle disableMe  
:disableMe SKIPPED  
  
BUILD SUCCESSFUL  
  
Total time: 1 secs
```

13.8. Task rules

Sometimes you want to have a task which behavior depends on a large or infinite number value range of parameters. A very nice and expressive way to provide such tasks are task rules:

Example 13.21. Task rule

build.gradle

```
tasks.addRule("Pattern: ping<ID>") { String taskName ->  
    if (taskName.startsWith("ping")) {  
        task(taskName) << {  
            println "Pinging: " + (taskName - 'ping')  
        }  
    }  
}
```

Output of `gradle -q pingServer1`

```
> gradle -q pingServer1  
Pinging: Server1
```

The String parameter is used as a description for the rule. This description is shown when doing for example `gradle -t`.

Rules not just work for calling tasks from the command line. You can also create `dependsOn` relations on rule based tasks:

Example 13.22. Dependency on rule based tasks

build.gradle

```
tasks.addRule("Pattern: ping<ID>") { String taskName ->
    if (taskName.startsWith("ping")) {
        task(taskName) << {
            println "Pinging: " + (taskName - 'ping')
        }
    }
}

task groupPing {
    dependsOn pingServer1, pingServer2
}
```

Output of `gradle -q groupPing`

```
> gradle -q groupPing
Pinging: Server1
Pinging: Server2
```

13.9. Summary

If you are coming from Ant, such an enhanced Gradle task as *Copy* looks like a mixture between an Ant target and an Ant task. And this is actually the case. The separation that Ant does between tasks and targets is not done by Gradle. The simple Gradle tasks are like Ant's targets and the enhanced Gradle tasks also include the Ant task aspects. All of Gradle's tasks share a common API and you can create dependencies between them. Such a task might be nicer to configure than an Ant task. It makes full use of the type system, is more expressive and easier to maintain.

[11] If you use the Java Plugin, this task is automatically created and added to your project.

[12] You might be wondering why there is neither an import for the `StopExecutionException` nor do we access it via its fully qualified name. The reason is, that Gradle adds a set of default imports to your script. These imports are customizable (see [Appendix C, Existing IDE Support and how to cope without it](#)).

Working With Files

This chapter is a work in progress.

Most builds work with files. Gradle adds some concepts and APIs to help you do this.

14.1. Locating files

You can locate a file relative to the project directory using the `Project.file()` method.

Example 14.1. Locating files

build.gradle

```
// Using a relative path
File configFile = file('src/config.xml')

// Using an absolute path
configFile = file(configFile.absolutePath)

// Using a File object with a relative path
configFile = file(new File('src/config.xml'))
```

You can pass any object to the `file()` method, though usually you would pass it a `String` or `File`. The supplied object's `toString()` value is treated as a file path. If this path is an absolute path, it is used to construct a `File` instance. Otherwise, a `File` instance is constructed by prepending the project directory path to the supplied path.

Using this method is a useful way to convert some user provided value into an absolute `File`. It is preferable to using `new File(somePath)`, as `file()` always evaluates the supplied path relative to the project directory, which is fixed, rather than the current working directory, which may not always be the same.

14.2. File collections

A *file collection* is simply a set of files. It is represented by the `FileCollection` interface. Many objects in Gradle implement this interface. For example, `dependency configurations` implement `FileCollection`.

One way to obtain a `FileCollection` instance is to use the `Project.files()` method. You can pass

this method any number of objects, which are then converted into a set of `File` objects. The `files()` method accepts `File` and `String` arguments. These are evaluated relative to the project directory, as described in [Section 14.1, “Locating files”](#). You can also pass collections, maps and arrays to the `files()` method. These are flattened and the contents converted to `File` instances.

Example 14.2. Creating a file collection

build.gradle

```
FileCollection collection = files('src/file1.txt', file('src/file2.txt'), ['src/file3.t
```

A file collection is iterable, and can be converted to a number of other types using the `as` operator. You can also add 2 file collections together using the `+` operator. Here are some examples.

Example 14.3. Using a file collection

build.gradle

```
// Iterate over the files in the collection
collection.each {File file ->
    file.name
}

// Convert the collection to various types
Set set = collection.files
Set set2 = collection as Set
List list = collection as List
String path = collection.asPath
File file = collection.singleFile
File file2 = collection as File

// Add collections together
def collection2 = collection + files('src/file3.txt')
```

You can also pass the `files()` method a closure or a `Callable` instance. This is called when the contents of the collection are queried, and its return value is converted to a set of `File` instances. The return value can be an object of any of the types supported by the `files()` method. This is a simple way to 'implement' the `FileCollection` interface.

Example 14.4. Implementing a file collection

build.gradle

```
task list << {
    File srcDir

    // Create a file collection using a closure
    collection = files { srcDir.listFiles() }

    // Create a file collection using a Callable
    Callable callable = { srcDir.listFiles() } as Callable
    collection = files(callable)

    srcDir = file('src')
    println "Contents of $srcDir.name"
    collection.each { println relativePath(it) }

    srcDir = file('src2')
    println "Contents of $srcDir.name"
    collection.each { println relativePath(it) }
}
```

Output of `gradle -q list`

```
> gradle -q list
Contents of src
src/dir1
src/file1.txt
Contents of src2
src2/dir1
src2/dir2
```

The `files()` method also accepts `FileCollection` instances.

14.3. File trees

A *file tree* is a hierarchy of files, such as a directory tree. It is represented by the `FileTree` interface. The `FileTree` interface extends `FileCollection`, so you can treat a file tree exactly the same way as you would a file collection. Several objects in Gradle implement the `FileTree` interface, such as `source sets`.

One way to obtain a `FileTree` instance is to use the `Project.fileTree()` method. This creates a `FileTree` instance defined with a base directory, and optionally some Ant-style include and exclude patterns.

Example 14.5. Creating a file tree

build.gradle

```
// Create a file tree with a base directory
def tree = fileTree(dir: 'src/main')

// Add include and exclude patterns to the tree
tree.include '**/*.java'
tree.exclude '**/Abstract*'

// Create a tree using path
tree = fileTree('src').include('**/*.java')

// Create a tree using closure
tree = fileTree {
    from 'src'
    include '**/*.java'
}

// Create a tree using a map
tree = fileTree(dir: 'src', includes: ['**/*.java'])
```

You use a file tree in the same way you use a file collection. You can also visit the contents of the tree, and select a subtree using Ant-style patterns:

Example 14.6. Using a file tree

build.gradle

```
// Iterate over the contents of a tree
tree.each {File file ->
    println file
}

// Filter a tree
def filtered = tree.matching {
    include 'org/gradle/api/**'
}

// Add trees together
def sum = tree + fileTree(dir: 'src/test')

// Visit the nodes of the tree
tree.visit {node ->
    println "$node.relativePath => $node.file"
}
```

14.4. Specifying a set of files

Many objects in Gradle have properties which accept a logical set of files. For example, the `Compile` task has a `source` property, which defines the source files to compile. You can set the value of this property using any of the types supported by the `files()` method, which we have seen above. This means you can set the property using, for example, a `File`, `String`, collection, `FileCollection` or even a closure. Here are some examples:

Example 14.7. Specifying a set of files

build.gradle

```
compile {
    source = file('src/main/java')
}

compile {
    source = 'src/main/java'
}

compile {
    source = ['src/main/java', '../shared/java']
}

compile {
    source = fileTree(dir: 'src/main/java').matching { include 'org/gradle/api/**' }
}

compile {
    source = {
        fileTree(dir: 'src/main/java') + fileTree(dir: '../shared/java')
    }
}
```

Usually, there is a method with the same name as the property, which appends to the set of files. Again, this method accepts any of the types supported by the `files()` method.

Example 14.8. Specifying a set of files

build.gradle

```
compile {
    source 'src/main/java', 'src/main/groovy'
    source file('../shared/java')
    source { file('src/test/').listFiles() }
}
```

14.5. Copying files

You can use the `Copy` task to copy files. The copy task allows you to filter the contents of the files as they are copied, and to map the files names.

Example 14.9. Copying files using the copy task

build.gradle

```
task copyTask(type: Copy) {
    from 'src/main/webapp'
    into 'build/explodedWar'
    include '**/*.html'
    include '**/*.jsp'
    include 'assets/**'
}
```

You can also use the `Project.copy()` method.

Example 14.10. Copying files using the copy() method

build.gradle

```
task copyMethod << {  
    copy {  
        from 'src/main/webapp'  
        into 'build/explodedWar'  
        include '**/*.html'  
        include '**/*.jsp'  
        include 'assets/**'  
    }  
}
```

15

Logging

The log is the main 'UI' of a build tool. If it is too verbose, real warnings and problems are easily hidden by this. On the other hand you need the relevant information for figuring out if things have gone wrong. Gradle defines 6 log levels, as shown in [Table 15.1, "Log levels"](#). There are two Gradle-specific log levels, in addition to the ones you might normally see. Those levels are *QUIET* and *LIFECYCLE*. The latter is the default, and is used to report build progress.

Table 15.1. Log levels

Level	Used for
ERROR	Error messages
QUIET	Important information messages
WARNING	Warning messages
LIFECYCLE	Progress information messages
INFO	Information messages
DEBUG	Debug messages

15.1. Choosing a log level

You can use the command line switches shown in [Table 15.2, "Log level command-line options"](#) to choose different log levels. In [Table 15.3, "Stacktrace command-line options"](#) you find the command line switches which affect stacktrace logging.

Table 15.2. Log level command-line options

Option	Outputs Log Levels
no logging options	LIFECYCLE and higher
-q	QUIET and higher
-i	INFO and higher
-d	DEBUG and higher (that is, all log messages)

Table 15.3. Stacktrace command-line options

Option	Meaning
No stacktrace options	No stacktraces are printed to the console in case of a build error (e.g. a compile error). Only in case of internal exceptions will stacktraces be printed. If the loglevel option <code>-d</code> is chosen, truncated stacktraces are always printed.
<code>-s</code>	Truncated stacktraces are printed. We recommend this over full stacktraces. Groovy full stacktraces are extremely verbose (Due to the underlying dynamic invocation mechanisms. Yet they usually do not contain relevant information for what has gone wrong in <i>your</i> code.)
<code>-f</code>	The full stacktraces are printed out.

15.2. Writing your own log messages

A simple option for logging in your build file is to write messages to standard output. Gradle redirects anything written to standard output to it's logging system at the `QUIET` log level.

Example 15.1. Using stdout to write log messages

build.gradle

```
println 'A message which is logged at QUIET level'
```

Gradle also provides a `logger` property to a build script, which is an instance of `Logger`. This interface extends the SLF4J `Logger` interface and adds a few Gradle specific methods to it. Below is an example of how this is used in the build script:

Example 15.2. Writing your own log messages

build.gradle

```
logger.quiet('An info log message which is always logged.')
logger.error('An error log message.')
logger.warn('A warning log message.')
logger.lifecycle('A lifecycle info log message.')
logger.info('An info log message.')
logger.debug('A debug log message.')
logger.trace('A trace log message.')
```

You can also hook into Gradle's logging system from within other classes used in the build (classes from the `src` directory for example). Simply use an SLF4J logger. You can use this logger the same way as you use the provided logger in the build script.

Example 15.3. Using SLF4J to write log messages

build.gradle

```
org.slf4j.Logger slf4jLogger = org.slf4j.LoggerFactory.getLogger('some-logger')
slf4jLogger.info('An info log message logged using SLF4j')
```

15.3. Logging from external tools and libraries

Internally, Gradle uses Ant and Ivy. Both have their own logging system. Gradle redirects their logging output into the Gradle logging system. There is a 1:1 mapping from the Ant/Ivy log levels to the Gradle log levels, except the Ant/Ivy `TRACE` log level, which is mapped to Gradle `DEBUG` log level. This means the default Gradle log level will not show any Ant/Ivy output unless it is an error or a warning.

There are many tools out there which still use standard output for logging. By default, Gradle redirects standard output to the `QUIET` log level and standard error to the `ERROR` level. This behavior is configurable. Gradle provides a couple of switches for this. To change the log level that standard out is redirected to when your build script gets evaluated, the project object offers a method called `Project.captureStandardOutput()`.

Example 15.4. Configuring standard output capture

build.gradle

```
captureStandardOutput LogLevel.INFO
println 'A message which is logged at INFO level'
```

To change the log level for standard out during task execution, tasks offer a method also with the name `Task.captureStandardOutput()`.

Example 15.5. Configuring standard output capture for a task

build.gradle

```
task logInfo << {
    println 'A task message which is logged at INFO level'
}
logInfo.captureStandardOutput(LogLevel.INFO)
```

Tasks and projects also offer a method `disableStandardOutputCapture()` which causes the standard output to be send to the default standard output. If you need more fine grained control on how standard output is redirected you can use the class `StandardOutputLogging`.

Gradle also provides integration with the Java Util Logging, Jakarta Commons Logging and Log4j logging toolkits. Any log messages which your build classes write using these logging toolkits will be redirected to Gradle's logging system.

Using Ant from Gradle

Gradle provides excellent integration with Ant. You can use individual Ant tasks or entire Ant builds in your Gradle builds. In fact, you will find that it's far easier and more powerful using Ant tasks in a Gradle build script, than it is to use Ant's XML format. You could even use Gradle simply as a powerful Ant task scripting tool.

Ant can be divided into two layers. The first layer is the Ant language. It provides the syntax for the `build.xml`, the handling of the targets, special constructs like `macrodefs`, and so on. In other words, everything except the Ant tasks and types. Gradle understands this language, and allows you to import your Ant `build.xml` directly into a Gradle project. You can then use the targets of your Ant build as if they were Gradle tasks.

The second layer of Ant is its wealth of Ant tasks and types, like `javac`, `copy` or `jar`. For this layer Gradle provides integration simply by relying on Groovy, and the fantastic `AntBuilder`.

Finally, since build scripts are Groovy scripts, you can always execute an Ant build as an external process. Your build script may contain statements like: `"ant clean compile".execute()`.^[13]

You can use Gradle's Ant integration as a path for migrating your build from Ant to Gradle. For example, you could start by importing your existing Ant build. Then you could move your dependency declarations from the Ant script to your build file. Finally, you could move your tasks across to your build file, or replace them with some of Gradle's plugins. This process can be done in parts over time, and you can have a working Gradle build during the entire process.

16.1. Using Ant tasks and types in your build

In your build script, a property called `ant` is provided by Gradle. This is a reference to an `AntBuilder` instance. This `AntBuilder` is used to access Ant tasks, types and properties from your build script. There is a very simple mapping from Ant's `build.xml` format to Groovy, which is explained below.

You execute an Ant task by calling a method on the `AntBuilder` instance. You use the task name as the method name. For example, you execute the Ant `echo` task by calling the `ant.echo()` method. The attributes of the Ant task are passed as Map parameters to the method. Below is an example which executes the `echo` task. Notice that we can also mix Groovy code and the Ant task markup. This can be extremely powerful.

Example 16.1. Using an Ant task

build.gradle

```
task hello << {
    String greeting = 'hello from Ant'
    ant.echo(message: greeting)
}
```

Output of `gradle hello`

```
> gradle hello
:hello
[ant:echo] hello from Ant

BUILD SUCCESSFUL

Total time: 1 secs
```

You pass nested text to an Ant task by passing it as a parameter of the task method call. In this example, we pass the message for the `echo` task as nested text:

Example 16.2. Passing nested text to an Ant task

build.gradle

```
task hello << {
    ant.echo('hello from Ant')
}
```

Output of `gradle hello`

```
> gradle hello
:hello
[ant:echo] hello from Ant

BUILD SUCCESSFUL

Total time: 1 secs
```

You pass nested elements to an Ant task inside a closure. Nested elements are defined in the same way as tasks, by calling a method with the same name as the element we want to define.

Example 16.3. Passing nested elements to an Ant task

build.gradle

```
task zip << {
    ant.zip(destfile: 'archive.zip') {
        fileset(dir: 'src') {
            include(name: '**.xml')
            exclude(name: '**.java')
        }
    }
}
```

You can access Ant types in the same way that you access tasks, using the name of the type as the method

name. The method call returns the Ant data type, which you can then use directly in your build script. In the following example, we create an Ant `path` object, then iterate over the contents of it.

Example 16.4. Using an Ant type

build.gradle

```
task list << {
    def path = ant.path {
        fileset(dir: 'libs', includes: '*.jar')
    }
    path.list().each {
        println it
    }
}
```

More information about `AntBuilder` can be found in 'Groovy in Action' 8.4 or at the [Groovy Wiki](#)

16.1.1. Using custom Ant tasks in your build

To make custom tasks available in your build, you use the `typedef` Ant task, just as you would in a `build.xml` file. You can then refer to the custom Ant task as you would a built-in Ant task.

Example 16.5. Using a custom Ant task

build.gradle

```
task check << {
    ant.taskdef(resource: 'checkstyletask.properties') {
        classpath {
            fileset(dir: 'libs', include: '*.jar')
        }
    }
    ant.checkstyle(config: 'checkstyle.xml') {
        fileset(dir: 'src')
    }
}
```

You can use Gradle's dependency management to assemble the classpath to use for the custom tasks. To do this, you need to define a custom configuration for the classpath, then add some dependencies to the configuration. This is described in more detail in [Section 28.3, "How to declare your dependencies"](#).

Example 16.6. Declaring the classpath for a custom Ant task

build.gradle

```
configurations {
    checkstyle
}

dependencies {
    checkstyle group: 'checkstyle', name: 'checkstyle', version: '5.0'
}
```

To use the classpath configuration, use the `asPath` property of the custom configuration.

Example 16.7. Using a custom Ant task and dependency management together

build.gradle

```
task check << {
    ant.taskdef(resource: 'checkstyletask.properties', classpath: configurations.checkstyle)
    ant.checkstyle(config: 'checkstyle.xml') {
        fileset(dir: 'src')
    }
}
```

16.2. Importing an Ant build

You can use the `ant.importBuild()` method to import an Ant build into your Gradle project. When you import an Ant build, each Ant target is treated as a Gradle task. This means you can manipulate and execute the Ant targets in exactly the same way as Gradle tasks.

Example 16.8. Importing an Ant build

build.gradle

```
ant.importBuild 'build.xml'
```

build.xml

```
<project>
  <target name="hello">
    <echo>Hello, from Ant</echo>
  </target>
</project>
```

Output of `gradle hello`

```
> gradle hello
:hello
[ant:echo] Hello, from Ant

BUILD SUCCESSFUL

Total time: 1 secs
```

You can add a task which depends on an Ant target:

Example 16.9. Task that depends on Ant target

build.gradle

```
ant.importBuild 'build.xml'

task intro(dependsOn: hello) << {
    println 'Hello, from Gradle'
}
```

Output of **gradle intro**

```
> gradle intro
:hello
[ant:echo] Hello, from Ant
:intro
Hello, from Gradle

BUILD SUCCESSFUL

Total time: 1 secs
```

Or, you can add behaviour to an Ant target:

Example 16.10. Adding behaviour to an Ant target

build.gradle

```
ant.importBuild 'build.xml'

hello << {
    println 'Hello, from Gradle'
}
```

Output of **gradle hello**

```
> gradle hello
:hello
[ant:echo] Hello, from Ant
Hello, from Gradle

BUILD SUCCESSFUL

Total time: 1 secs
```

It is also possible for an Ant target to depend on a Gradle task:

Example 16.11. Ant target that depends on Gradle task

build.gradle

```
ant.importBuild 'build.xml'

task intro << {
    println 'Hello, from Gradle'
}
```

build.xml

```
<project>
    <target name="hello" depends="intro">
        <echo>Hello, from Ant</echo>
    </target>
</project>
```

Output of `gradle hello`

```
> gradle hello
:intro
Hello, from Gradle
:hello
[ant:echo] Hello, from Ant

BUILD SUCCESSFUL

Total time: 1 secs
```

16.3. Ant properties and references

There are several ways to set an Ant property, so that the property can be used by Ant tasks. You can set the property directly on the `AntBuilder` instance. The Ant properties are also available as a `Map` which you can change. You can also use the `Ant property` task. Below are some examples of how to do this.

Example 16.12. Setting an Ant property

build.gradle

```
ant.buildDir = buildDir
ant.properties.buildDir = buildDir
ant.properties['buildDir'] = buildDir
ant.property(name: 'buildDir', location: buildDir)
```

build.xml

```
<echo>buildDir = ${buildDir}</echo>
```

Many Ant tasks set properties when they execute. There are several ways to get the value of these properties. You can get the property directly from the `AntBuilder` instance. The Ant properties are also available as a `Map`. Below are some examples.

Example 16.13. Getting an Ant property

build.xml

```
<property name="antProp" value="a property defined in an Ant build"/>
```

build.gradle

```
println ant.antProp
println ant.properties.antProp
println ant.properties['antProp']
```

There are several ways to set an Ant reference:

Example 16.14. Setting an Ant reference

build.gradle

```
ant.path(id: 'classpath', location: 'libs')
ant.references.classpath = ant.path(location: 'libs')
ant.references['classpath'] = ant.path(location: 'libs')
```

build.xml

```
<path refid="classpath"/>
```

There are several ways to get an Ant reference:

Example 16.15. Getting an Ant reference

build.xml

```
<path id="antPath" location="libs"/>
```

build.gradle

```
println ant.references.antPath
println ant.references['antPath']
```

16.4. API

The Ant integration is provided by `AntBuilder`.

[13] In Groovy you can execute Strings. To learn more about executing external processes with Groovy have a look in 'Groovy in Action' 9.3.2 or at the Groovy wiki

17

Using Plugins

Now we look at *how* Gradle provides build-by-convention and out of the box functionality. These features are decoupled from the core of Gradle, and are provided via plugins. Although the plugins are decoupled, we would like to point out that the Gradle core plugins are **NEVER** updated or changed for a particular Gradle distribution. If there is a bug in the compile functionality of Gradle, we will release a new version of Gradle. There is no change of behavior for the lifetime of a given distribution of Gradle.

17.1. Declaring plugins

If you want to use the plugin for building a Java project, simply include

```
usePlugin('java')
```

in your script. That's all. From a technological point of view plugins use just the same operations as you can use from your build scripts. That is, they use the Project and Task API (see [Chapter 12, *The Project and Task API*](#)). The Gradle plugins use this API to:

- Add tasks to the project (e.g. compile, test)
- Create dependencies between those tasks to let them execute in the appropriate order.
- Add dependency configurations to the project.
- Add a so called *convention object* to the project.

Let's check this out:

Example 17.1. Using plugin

build.gradle

```
usePlugin 'java'

task show << {
    println relativePath(compileJava.destinationDir)
    println relativePath(processResources.destinationDir)
}
```

Output of **gradle -q show**

```
> gradle -q show
build/classes/main
build/classes/main
```

The Java Plugin adds a `compileJava` task and a `processResources` task to the project object which can be accessed by a build script. It has configured the `destinationDir` property of both of these tasks.

The `usePlugin()` method either takes a string or a class as an argument. You can write ^[14]

```
usePlugin(org.gradle.api.plugins.JavaPlugin)
```

Any class, which implements the `Plugin` interface, can be used as a plugin. Just pass the class as an argument. You don't need to configure anything else for this. If you want to access a custom plugin via a string identifier, you must inform Gradle about the mapping. You can do this in the file `plugin.properties` in the top level directory of Gradle. It looks like this for the current release:

Figure 17.1. plugin.properties

```
#
# Copyright 2009 the original author or authors.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#
base=org.gradle.api.plugins.BasePlugin
java=org.gradle.api.plugins.JavaPlugin
eclipse=org.gradle.api.plugins.EclipsePlugin
groovy=org.gradle.api.plugins.GroovyPlugin
war=org.gradle.api.plugins.WarPlugin
osgi=org.gradle.api.plugins.osgi.OsgiPlugin
jetty=org.gradle.api.plugins.jetty.JettyPlugin
maven=org.gradle.api.plugins.MavenPlugin
project-reports=org.gradle.api.plugins.ProjectReportsPlugin
code-quality=org.gradle.api.plugins.quality.CodeQualityPlugin
scala=org.gradle.api.plugins.scala.ScalaPlugin
```

If you want to use your own plugins, you must make sure that they are accessible via the build script classpath (see [Chapter 34, *Organizing Build Logic*](#) for more information). To learn more about how to write custom plugins, see [Chapter 33, *Writing Custom Plugins*](#).

17.2. Using the convention object

If you use the Java Plugin for example, there are a `compileJava` and a `processResources` task for your production code (the same is true for your test code). The default location for the output of those tasks is the directory `build/classes/main`. What if you want to change this? Let's try:

Example 17.2. Configuring a plugin task

build.gradle

```
usePlugin 'java'

task show << {
    processResources.destinationDir = new File(buildDir, 'output')
    println relativePath(processResources.destinationDir)
    println relativePath(compileJava.destinationDir)
}
```

Output of `gradle -q show`

```
> gradle -q show
build/output
build/classes/main
```

Setting the `destinationDir` of the `processResources` task had only an effect on the `processResources` task. Maybe this was what you wanted. But what if you want to change the output directory for all tasks? It would be unfortunate if you had to do this for each task separately.

Gradle's tasks are usually *convention aware*. A plugin can add a convention object to your project, and map certain values of this convention object to task properties.

Example 17.3. Plugin convention object

build.gradle

```
usePlugin 'java'

task show << {
    sourceSets.main.classesDir = new File(buildDir, 'output')
    println relativePath(processResources.destinationDir)
    println relativePath(compileJava.destinationDir)
}
```

Output of `gradle -q show`

```
> gradle -q show
build/output
build/output
```

The Java Plugin has added a convention object with a `sourceSets` property, which we use to set the classes directory. Notice that setting this has changed the `destinationDir` property of both the `processResources`

and `compileJava` tasks.

By setting a task attribute explicitly (as we have done in the first example) you overwrite the convention value for this particular task.

Not all of the tasks attributes are mapped to convention object values. It is the decision of the plugin to decide what are the shared properties and then bundle them in a convention object and map them to the tasks.

The properties of a convention object can be accessed as project properties. As shown in the following example, you can also access the convention object explicitly.

Example 17.4. Using the plugin convention object

build.gradle

```
usePlugin 'java'

task show << {
    // Access the convention property as a project property
    println relativePath(sourceSets.main.classesDir)
    println relativePath(project.sourceSets.main.classesDir)

    // Access the convention property via the convention object
    println relativePath(convention.sourceSets.main.classesDir)
    println relativePath(convention.plugins.java.sourceSets.main.classesDir)
}
```

Output of `gradle -q show`

```
> gradle -q show
build/classes/main
build/classes/main
build/classes/main
build/classes/main
```

Every project object has a `Convention` object which is a container for convention objects contributed by the plugins declared for your project. If you simply access or set a property or access a method in your build script, the project object first looks if this is a property of itself. If not, it delegates the request to its convention object. The convention object checks if any of the plugin convention objects can fulfill the request (first wins and the order is not defined). The plugin convention objects also introduce a namespace.

17.2.1. Declaring plugins multiple times

A plugin is only called once for a given project, even if you have multiple `usePlugin()` statements. An additional call after the first call has no effect but doesn't hurt either. This can be important if you use plugins which extend other plugins. For example `usePlugin('groovy')` calls also the Java Plugin. We say the Groovy plugin extends the Java plugin. But you might as well write:

```
usePlugin('java')
usePlugin('groovy')
```

If you use cross-project configuration in multi-project builds this is a useful feature.

17.3. Summary

Plugins provide tasks, which are glued together via `dependsOn` relations and a convention object.

[14] Thanks to Gradle's default imports (see [Appendix C, Existing IDE Support and how to cope without it](#)) you can also write `usePlugin(JavaPlugin)` in this case.

18

The Java Plugin

The Java plugin adds Java compilation, testing and bundling capabilities to a project. It serves as the basis for many of the other Gradle plugins.

18.1. Source sets

The Java plugin introduces the concept of a *source set*. A source set is a group of source files which are compiled and executed together. These source files may include Java source files and resource files. Other plugins add the ability to include Groovy and Scala source files in a source set. A source set has an associated compile classpath, and runtime classpath.

You might use a source set to define an integration test suite, or for the API classes of your project, or to separate source which needs to be compiled against different Java versions.

The Java plugin defines two standard source sets, called `main` and `test`. The `main` source set contains your production source code, which is compiled and assembled into a JAR file. The `test` source set contains your unit test source code, which is compiled and executed using JUnit or TestNG.

18.2. Tasks

The Java plugin adds a number of tasks to your project, as shown below.

Table 18.1. Java plugin - tasks

Task name	Depends on	Type	Description
<code>clean</code>	-	<u>Clean</u>	Deletes the project build directory.
<code>compileJava</code>	All tasks which produce the compile classpath. This includes the <code>jar</code> task for project dependencies included in the <code>compile</code> configuration.	<u>Compile</u>	Compiles production Java source files using <code>javac</code> .

<code>processResources</code>	-	<u>Copy</u>	Copies production resources into the production classes directory.
<code>classes</code>	<code>compileJava</code> and <code>processResources</code> . Some plugins add additional compilation tasks.	<u>Task</u>	Assembles the production classes directory.
<code>compileTestJava</code>	<code>compile</code> , plus all tasks which produce the test compile classpath.	<u>Compile</u>	Compiles test Java source files using <code>javac</code> .
<code>processTestResources</code>	-	<u>Copy</u>	Copies test resources into the test classes directory.
<code>testClasses</code>	<code>compileTestJava</code> and <code>processTestResources</code> . Some plugins add additional test compilation tasks.	<u>Task</u>	Assembles the test classes directory.
<code>jar</code>	<code>compile</code>	<u>Jar</u>	Assembles the JAR file
<code>javadoc</code>	<code>compile</code>	<u>Javadoc</u>	Generates API documentation for the production Java source, using Javadoc
<code>test</code>	<code>compile</code> , <code>compileTest</code> , plus all tasks which produce the test runtime classpath.	<u>Test</u>	Runs the unit tests using JUnit or TestNG.
<code>uploadArchives</code>	The tasks which produce the artifacts in the <code>archives</code> configuration, including <code>jar</code> .	<u>Upload</u>	Uploads the artifacts in the <code>archives</code> configuration, including the JAR file.

For each source set you add to the project, the Java plugin adds the following compilation tasks:

Table 18.2. Java plugin - source set tasks

Task name	Depends on	Type	Description
<code>compileSourceSetClasses</code>	All tasks which produce the source set's compile classpath.	<u>Compile</u>	Compiles the given source set's Java source files using javac.
<code>processSourceSetResources</code>		<u>Copy</u>	Copies the given source set's resources into the classes directory.
<code>sourceSetClassesTask</code>	<code>compileSourceSetJava</code> and <code>processSourceSetResources</code> . Some plugins add additional compilation tasks for the source set.	<u>Task</u>	Assembles the given source set's classes directory.

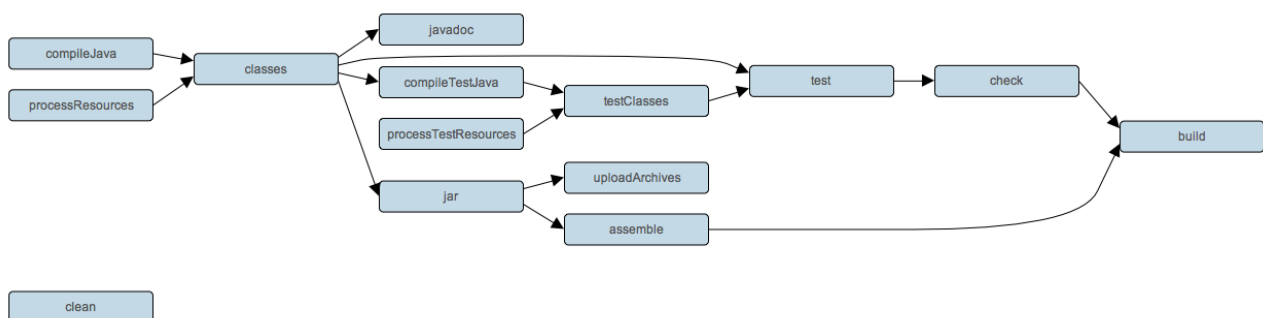
The Java plugin also adds a number of tasks which form a lifecycle for the project:

Table 18.3. Java plugin - lifecycle tasks

Task name	Depends on	Type	Description
assemble	All archive tasks in the project, including <code>jar</code> . Some plugins add additional archive tasks to the project.	Task	Assembles all the archives in the project.
check	All verification tasks in the project, including <code>test</code> . Some plugins add additional verification tasks to the project.	Task	Performs all verification tasks in the project.
build	<code>check</code> and <code>assemble</code>	Task	Performs a full build of the project.
buildNeeded	<code>build</code> and <code>build</code> tasks in all project lib dependencies of the <code>testRuntime</code> configuration.	Task	Performs a full build of the project and all projects it depends on.
buildDependents	<code>build</code> and <code>build</code> tasks in all projects with a project lib dependency on this project in a <code>testRuntime</code> configuration.	Task	Performs a full build of the project and all projects which depend on it.
<code>buildConfigurationName</code>	The tasks which produce the artifacts in configuration <code>ConfigurationName</code> .	Task	Assembles the artifacts in the specified configuration.
<code>uploadConfigurationName</code>	The tasks which uploads the artifacts in configuration <code>ConfigurationName</code> .	Upload	Assembles and uploads the artifacts in the specified configuration.

The following diagram shows the relationships between these tasks.

Figure 18.1. Java plugin - tasks



18.3. Project layout

The Java plugin assumes the project layout shown below. None of these directories need exist or have anything in them. The Java plugin will compile whatever it finds, and handles anything which is missing.

Table 18.4. Java plugin - default project layout

Directory	Meaning
<code>src/main/java</code>	Production Java source
<code>src/main/resources</code>	Production resources
<code>src/test/java</code>	Test Java source
<code>src/test/resources</code>	Test resources
<code>src/sourceSet/java</code>	Java source for the given source set
<code>src/sourceSet/resources</code>	Resources for the given source set

18.3.1. Changing the project layout

You configure the project layout by configuring the appropriate source set. This is discussed in more detail in the following sections. Here is a brief example which changes the main Java and resource source directories.

Example 18.1. Custom Java source layout

build.gradle

```
sourceSets {
    main {
        java {
            srcDir 'src/java'
        }
        resources {
            srcDir 'src/resources'
        }
    }
}
```

18.4. Dependency management

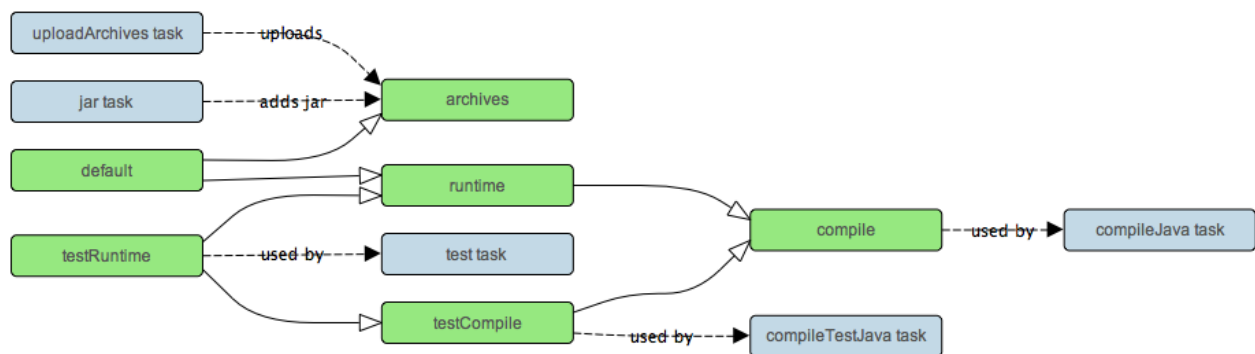
The Java plugin adds a number of dependency configurations to your project, as shown below. It assigns those configurations to tasks such as `compileJava` and `test`. To learn more about configurations see [Section 28.3.1, “Configurations”](#) and [Section 29.2, “Artifacts and configurations”](#). Note also that transitive dependencies are disabled by default for the compile configuration. This can be overridden using:

```
configurations.compile.transitive = true
```

Table 18.5. Java plugin - dependency configurations

Name	Extends	Used by tasks	Meaning
compile	-	compileJava	Compile time dependencies
runtime	compile	-	Runtime dependencies
testCompile	compile	compileTestJava	Additional dependencies for compiling tests.
testRuntime	runtime, testCompile	test	Additional dependencies for running tests only.
archives	-	uploadArchives	Artifacts (e.g. jars) produced by this project.
default	runtime, archives	-	Artifacts produced and dependencies required by this project.

Figure 18.2. Java plugin - dependency configurations



18.5. Convention properties

The Java plugin adds a number of convention properties to the project, shown below. You can use these properties in your build script as though they were properties of the project object (see [Section 17.2, “Using the convention object”](#)).

Table 18.6. Java plugin - directory properties

Property name	Type	Default value	Description
reportsDirName	String	reports	The name of the directory to generate reports into, relative to the build directory.
reportsDir	File (read-only)	<i>buildDir/reportsDirName</i>	The directory to generate reports into.
testResultsDirName	String	test-results	The name of the directory to generate test result .xml files into, relative to the build directory.

testResultsDir	File (read-only)	<i>buildDir/testResultsDirName</i>	The directory to generate test result .xml files into.
testReportDirName	String	tests	The name of the directory to generate the test report into, relative to the reports directory.
testReportDir	File (read-only)	<i>reportsDir/testReportDirName</i>	The directory to generate the test report into.
libsDirName	String	libs	The name of the directory to generate libraries into, relative to the build directory.
libsDir	File (read-only)	<i>buildDir/libsDirName</i>	The directory to generate libraries into.
distsDirName	String	dists	The name of the directory to generate distributions into, relative to the build directory.
distsDir	File (read-only)	<i>buildDir/distsDirName</i>	The directory to generate distributions into.
docsDirName	String	docs	The name of the directory to generate documentation into, relative to the build directory.
docsDir	File (read-only)	<i>buildDir/docsDirName</i>	The directory to generate documentation into.
dependencyCacheDirName	String	dependency-cache	The name of the directory to use to cache source dependency information, relative to the build directory.
dependencyCacheDir	File (read-only)	<i>buildDir/dependencyCacheDirName</i>	The directory to use to cache source dependency information.

Table 18.7. Java plugin - other properties

Property name	Type	Default value	Description
<code>sourceSets</code>	<code>SourceSetContainer</code> (read-only)	Not null	Contains the project's source sets.
<code>sourceCompatibility</code>	<code>JavaVersion</code> . Can also set using a String or a Number, eg '1.5' or 1.5.	1.5	Java version compatibility to use when compiling Java source.
<code>targetCompatibility</code>	<code>JavaVersion</code> . Can also set using a String or Number, eg '1.5' or 1.5.	<code>sourceCompatibility</code>	Java version to generate classes for.
<code>archivesBaseName</code>	String	<code>projectName</code>	The basename to use for archives, such as JAR or ZIP files.
<code>manifest</code>	<code>GradleManifest</code>	an empty manifest	The manifest to include in all JAR files.
<code>metaInf</code>	List	[]	A set of file collections which specify the files to include in the META-INF directory or all JAR files.

These properties are provided by convention objects of type `JavaPluginConvention`, `BasePluginConvention` and `ReportingBasePluginConvention`.

18.6. Working with source sets

You can access the source sets of a project using the `sourceSets` property. This is a container for the project's source sets, of type `SourceSetContainer`. There is also a `sourceSets()` method, which you can pass a closure to which configures the source set container. The source set container works pretty much the same way as other containers, such as `tasks`.

Example 18.2. Accessing a source set

build.gradle

```
// Various ways to access the main source set
println sourceSets.main.classesDir
println sourceSets['main'].classesDir
sourceSets {
    println main.classesDir
}
sourceSets {
    main {
        println classesDir
    }
}

// Iterate over the source sets
sourceSets.each {SourceSet set ->
    println set.name
}
```

To configure an existing source set, you simply use one of the above access methods to set the properties of the source set. The properties are described below. Here is an example which configures the main Java and resources directories:

Example 18.3. Configuring the source directories of a source set

build.gradle

```
sourceSets {
    main {
        java {
            srcDir 'src/java'
        }
        resources {
            srcDir 'src/resources'
        }
    }
}
```

To define a new source set, you simply reference it in the `sourceSets { }` block. When you define a source set, the Java plugin adds a number of tasks which assemble the classes for the source set, as shown in [Table 18.2, “Java plugin - source set tasks”](#). For example, if you add a source set called `intTest`, the Java plugin adds `compileIntTestJava`, `processIntTestResources` and `intTestClasses` tasks.

Example 18.4. Defining a source set

build.gradle

```
sourceSets {
    intTest
}
```

18.6.1. Source set properties

The following table lists some of the important properties of a source set. You can find more details in the API documentation for [SourceSet](#).

Table 18.8. Java plugin - source set properties

Property name	Type	Default value	Description
name	String (read-only)	Not null	The name of the source set, used to identify it.
classesDir	File	<i>buildDir/classes/name</i>	The directory to generate the classes of this source set into.
compileClasspath	<u>FileCollection</u>	compile Configuration.	The classpath to use when compiling the source files of this source set.
runtimeClasspath	<u>FileCollection</u>	classesDir + runtime Configuration.	The classpath to use when executing the classes of this source set.
java	<u>SourceDirectorySet</u> (read-only)	Not null	The Java source files of this source set. Contains only . java files found in the Java source directories, and excludes all other files.
java.srcDirs	Set<File>. Can set using anything described in <u>Section 14.4, “Specifying a set of files”</u> .	<i>[projectDir/src/name]</i>	The source directories containing the Java source files of this source set.
resources	<u>SourceDirectorySet</u> (read-only)	Not null	The resources of this source set. Contains only resources, and excludes any . java files found in the resource source directories. Other plugins, such as the Groovy plugin, exclude additional types of files from this collection.

<code>resources.srcDirs</code>	<code>Set<File></code> . Can set using anything described in Section 14.4 , “ Specifying a set of files ”.	<code>[projectDir/src/main/resources]</code>	The source directories containing the resources of this source set.
<code>allJava</code>	<code>FileTree</code> (read-only)	<code>java</code>	All <code>.java</code> files of this source set. Some plugins, such as the Groovy plugin, add additional Java source files to this collection.
<code>allSource</code>	<code>FileTree</code> (read-only)	<code>resources + java</code>	All source files of this source set. This include all resource files and all Java source files. Some plugins, such as the Groovy plugin, add additional source files to this collection.

18.6.2. Some common source set examples

Using dependency configurations to define the source set classpath:

Example 18.5. Defining the classpath of a source set

`build.gradle`

```

configurations {
    intTestCompile { extendsFrom compile }
    intTestRuntime { extendsFrom intTestCompile, runtime }
}

sourceSets {
    intTest {
        compileClasspath = sourceSets.main.classes + configurations.intTestCompile
        runtimeClasspath = classes + sourceSets.main.classes + configurations.intTestRuntime
    }
}

```

Adding a JAR containing the classes of a source set:

Example 18.6. Assembling a JAR for a source set

`build.gradle`

```

task intTestJar(type: Jar) {
    from sourceSets.intTest.classes
}

```

Generating Javadoc for a source set:

Example 18.7. Generating the Javadoc for a source set

build.gradle

```
task intTestJavadoc(type: Javadoc) {
    source sourceSets.intTest.allJava
}
```

Adding a test suite to run the tests in a source set:

Example 18.8. Running tests in a source set

build.gradle

```
task intTest(type: Test) {
    testClassesDir = sourceSets.intTest.classesDir
    classpath = sourceSets.intTest.runtimeClasspath
}
```

18.7. Javadoc

The `javadoc` task is an instance of `Javadoc`. It supports the core javadoc options and the options of the standard doclet described in the [reference documentation](#) of the Javadoc executable. For a complete list of supported Javadoc options consult the API documentation of the following classes: `CoreJavadocOptions` and `StandardJavadocDocletOptions`.

Table 18.9. Java plugin - Javadoc properties

Task Property	Type	Default Value
classpath	FileCollection	<code>sourceSets.main.classes + sourceSets.main.compileClasspath</code>
source	FileTree . Can set using anything described in Section 14.4 , “Specifying a set of files”.	<code>sourceSets.main.allJava</code>
destinationDir	<code>File</code>	<code>docsDir/javadoc</code>
title	<code>String</code>	The name and version of the project

18.8. Clean

The `clean` task is an instance of `Clean`. It simply removes the directory denoted by its `dir` property.

Table 18.10. Java plugin - Clean properties

Task Property	Type	Default Value
<code>dir</code>	<code>File</code>	<code>buildDir</code>

18.9. Resources

The Java plugin uses the `Copy` task for resource handling. It adds an instance for each source set in the project. You can find out more about the copy task in [Section 14.5, “Copying files”](#).

Table 18.11. Java plugin - ProcessResources properties

Task Property	Type	Default Value
<code>srcDirs</code>	Object. Can set using anything described in Section 14.4, “Specifying a set of files” .	<code>sourceSet.resou</code>
<code>destinationDir</code>	File. Can set using anything described in Section 14.1, “Locating files” .	<code>sourceSet.class</code>

18.10. CompileJava

The Java plugin adds a `Compile` instance for each source set in the project. The compile task delegates to Ant's `javac` task to do the compile. You can set most of the properties of the Ant `javac` task.

Table 18.12. Java plugin - Compile properties

Task Property	Type	Default Value
<code>classpath</code>	<code>FileCollection</code>	<code>sourceSet.compileC</code>
<code>source</code>	<code>FileTree</code> . Can set using anything described in Section 14.4, “Specifying a set of files” .	<code>sourceSet.java</code>
<code>destinationDir</code>	File.	<code>sourceSet.classesD</code>

18.11. Test

The `test` task is an instance of `Test`. It executes all unit tests found in the `test` source set.

Table 18.13. Java plugin - test properties

Task Property	Type	Default Value
<code>testClassesDir</code>	File	<code>sourceSets.test.classesDir</code>
<code>classpath</code>	<code>FileCollection</code>	<code>sourceSets.test.runtimeClasspath</code>
<code>testResultsDir</code>	File	<code>testResultsDir</code>
<code>testReportDir</code>	File	<code>testReportDir</code>
<code>testSrcDirs</code>	<code>List<File></code>	<code>sourceSets.test.java.srcDirs</code>

Have a look at `Test` for its complete API. Right now the test results are always in XML-format. The task has a `stopAtFailuresOrErrors` property to control the behavior when tests are failing. Test *a/ways* executes all tests. It stops the build afterwards if `stopAtFailuresOrErrors` is true and there are failing tests or tests that have thrown an uncaught exception.

Per default the tests are run in a forked JVM and the fork is done per test. You can modify this behavior by

setting forking to false or set the forkmode to once.

The Test task detects which classes are test classes by inspecting the compiled test classes. By default it scans all `.class` files. You can set custom includes / excludes, only those classes will be scanned. Depending on the Test framework used (JUnit / TestNG) the test class detection uses different criteria.

When using JUnit, we scan for both JUnit 3 and 4 test classes. If any of the following criteria match, the class is considered to be a JUnit test class. Extend `TestCase` or `GroovyTestCase`, Class annotated with `RunWith` or contain a method annotated with `Test` (inherited test methods are detected).

When using TestNG, we scan for methods annotated with `Test` (inherited test methods are detected).

Since 0.6.1 we scan up the inheritance tree into jar files on the test classpath.

In case you don't want to use the test class detection, you can disable it by setting `scanForTestClasses` to false. This will make the test task only use the includes / excludes to find test classes. If `scanForTestClasses` is disabled and no include or exclude patterns are specified, the respective defaults are used. For include this is `"/**/*Tests.class"`, `"/**/*Test.class"` and the for exclude it is `"/Ab;`
.

Both JUnit and TestNG are supported through their Ant tasks.

Regarding TestNG reporting, when the test report is disabled the default TestNG listeners are disabled (`options.useDefaultListeners` is set to false).

18.12. Jar

The `jar` task creates a JAR file containing the class files and resources of the project. The JAR file is declared as an artifact in the `archives` dependency configuration. This means that the JAR is available in the classpath of a dependent project. If you upload your project into a repository, this JAR is declared as part of the dependency descriptor. To learn more about how to work with archives and artifact configurations see [Chapter 29, *Artifact Management*](#).

18.13. Adding archives

If you come from Maven you can have only one library JAR per project. With Gradle you can have as many as you want. You can also add WAR, ZIP and TAR archives to your project. They are all added the same way, so let's look at how you add a ZIP file.

Example 18.9. Creation of ZIP archive

build.gradle

```
usePlugin 'java'
version = 1.0

task myZip(type: Zip) {
    fileSet(dir: 'somedir')
}

println myZip.archiveName
```

Output of **gradle -q myZip**

```
> gradle -q myZip
zipProject-1.0.zip
```

This adds a Zip archive task with the name `myZip` which produces ZIP file `zipProject-1.0.zip`. It is important to distinguish between the name of the archive task and the name of the archive generated by the archive task. The name of the generated archive file is by default the name of the project with the project version appended. The default name for archives can be changed with the `archivesBaseName` project property. The name of the archive can also be changed at any time later on.

There are a number of properties which you can set on an archive task. You can, for example, change the name of the archive:

Example 18.10. Configuration of archive task - custom archive name

build.gradle

```
usePlugin 'java'
version = 1.0

task myZip(type: Zip) {
    fileSet(dir: 'somedir')
    baseName = 'customName'
}

println myZip.archiveName
```

Output of **gradle -q myZip**

```
> gradle -q myZip
customName-1.0.zip
```

You can further customize the archive names:

Example 18.11. Configuration of archive task - appendix & classifier

build.gradle

```
usePlugin 'java'
archivesBaseName = 'gradle'
version = 1.0

task myZip(type: Zip) {
    appendix = 'wrapper'
    classifier = 'src'
    fileSet(dir: 'somedir')
}

println myZip.archiveName
```

Output of `gradle -q myZip`

```
> gradle -q myZip
gradle-wrapper-1.0-src.zip
```

Often you will want to publish an archive, so that it is usable from another project. This process is described in [Chapter 29, *Artifact Management*](#)

18.13.1. Archive tasks

An archive task is a task which produces an archive at execution time. The following archive tasks are available:

Table 18.14. Archive tasks

Type	Accepted file container	Extends
<u>Zip</u>	fileSet, fileCollection, zipFileSet	AbstractArchiveTask
<u>Tar</u>	fileSet, fileCollection, zipFileSet, tarFileSet	Zip
<u>Jar</u>	fileSet, fileCollection, zipFileSet	Zip
<u>War</u>	fileSet, fileCollection, zipFileSet	Jar

The following file containers are available:

Table 18.15. File container for archives

Type	Meaning
FileTree	A set of files defined by a common base directory and include/exclude patterns. See Section 14.3, “File trees” .
ZipFileSet	Extends FileSet with additional properties known from Ant's zipfileset task.
TarFileSet	Extends ZipFileSet with additional properties known from Ant's tarfileset task.
FileCollection	An arbitrary collection of files to include in the archive. In contrast to a FileTree they don't need to have a common base directory. See Section 14.2, “File collections” for more details.
AntDirective	An arbitrary Ant resource declaration.

To learn about all the details have a look at the javadoc of the archive task class or the file container class itself.

18.13.1.1. Common properties

The name of the generated archive is assembled from the task properties `baseName`, `appendix`, `version`, `classifier` and `extension` into `baseName-appendix-version-classifier.extension`.^[15] The assembled name is accessible via the `archiveName` property. The `name` property denotes the name of the task, not the generated archive. An archive task has also a `customName` property. If this property is set, the `name` property returns its value instead of assembling a name out of the properties mentioned above.

Archives have a `destinationDir` property to specify where the generated archive should be placed. It has also an `archivePath` property, which returns a File object with the absolute path of the generated archive.

18.13.1.2. Adding content

To add content to an archive you must add file container to an archive (see [Table 18.15, “File container for archives”](#)). You can add as many file containers as you like. They behave pretty much the same as the Ant resources with similar names.

Example 18.12. Adding content to archive - include & exclude

build.gradle

```
task zipWithFileSet(type: Zip) {
    fileSet(dir: 'contentDir') {
        include('**/*.txt')
        exclude('**/*.gif')
    }
}
```

You can add arbitrary files to an archive:

Example 18.13. Adding content to archive - arbitrary files

build.gradle

```
task zipWithFiles(type: Zip) {
    files('path_to_file1', 'path_to_file2')
}
```

Other examples:

Example 18.14. Adding content to archive - zipFileSet

build.gradle

```
task zipWithZipFileSet(type: Zip) {
    zipFileSet(dir: 'contentDir') {
        include('**/*.txt')
        exclude('**/*.gif')
        prefix = 'myprefix'
    }
}
```

Example 18.15. Creation of TAR archive

build.gradle

```
task tarWithFileSet(type: Tar) {
    tarFileSet(dir: 'contentDir') {
        include('**/*.txt')
        exclude('**/*.gif')
        uid = 'myuid'
    }
}
```

There is also the option to add an arbitrary Ant expression describing an Ant resource.

```
myZipTask.antDirective {
    zipgroupfileset(dir: new File(rootDir, 'lib'))
}
```

This is for rather exotic use cases. Usually you should be fine with the file container provided by Gradle.

18.13.1.3. Merging

If you want to merge the content of other archives into the archive to be generated Gradle offers you two methods. One is merge:

```
myZipTask.merge('path1/otherArchive1.zip', 'path2/otherArchive.tar.gz')
```

This merges the whole content of the archive passed to the merge method into the generated archive. If you need more control which content of the archive should be merged and to what path, you can pass a closure to the merge method:

```
myZipTask.merge('path1/otherArchive1.zip', 'path2/otherArchive.tar.gz') {
    include('**/*.txt')
    exclude('**/*.gif')
    prefix = 'myprefix'
}
```

Under the hood Gradle scans the extension of the archives to be merged. According to the extension, it creates a `ZipFileSet` or `TarFileSet`. The closure is applied to this newly created file container. There is another method for merging called `mergeGroup`.

```
myZipTask.mergeGroup('path_to_dir_with_archives') {
    include('**/*.zip')
    exclude('**/*.tar.gz')
}
```

With this method you can assign a set of archives to be merged. Those archives have to be located under the directory you pass as an argument. You can define filters what archives should be included. They are always included fully and you can't specify a path. If you need this features, you must use the `merge` method.

18.13.1.4. Manifest

The convention object of the Java Plugin has a `manifest` property pointing to an instance of `GradleManifest`. With this `GradleManifest` object you can define the content of the `MANIFEST.MF` file for all the jar or a war archives in your project.

Example 18.16. Customization of MANIFEST.MF

build.gradle

```
manifest.mainAttributes("Implementation-Title": "Gradle", "Implementation-Version": ver
```

You can also define sections of a manifest file.

If a particular archive needs unique entries in its manifest you have to create your own `GradleManifest` instance for it.

Example 18.17. Customization of MANIFEST.MF for a particular archive

build.gradle

```
manifest.mainAttributes("Implementation-Title": "Gradle", "Implementation-Version": ver
myZipTask.manifest = new GradleManifest(manifest.createManifest())
myZipTask.manifest.mainAttributes(mykey: "myvalue")
```

Passing the common manifest object to the constructor of `GradleManifest` add the common manifest values to the task specific manifest instance.

18.13.1.5. MetaInf

The convention object of the Java Plugin has a `metaInf` property pointing to a list of `FileSet` objects. With these file sets you can define which files should be in the `META-INF` directory of a JAR or a WAR archive.

```
metaInf << new FileSet(someDir)
```

18.14. Uploading

How to upload your archives is described in [Chapter 29, *Artifact Management*](#).

18.15. Eclipse

Gradle comes with a number of tasks for generating eclipse files for your projects.

18.15.1. Eclipse classpath

`EclipseClasspath` has a default instance with the name `eclipseCp`. It generates a `.classpath` file.

Table 18.16. Java plugin - Eclipse properties

Task Property	Convention Property
<code>srcDirs</code>	<code>srcDirs + resourcesDirs</code>
<code>testSrcDirs</code>	<code>testSrcDirs + testResourcesDirs</code>
<code>outputDirectory</code>	<code>classesDir</code>
<code>testOutputDirectory</code>	<code>testClassesDir</code>
<code>classpathLibs</code>	the resolve result for <code>testRuntime</code>

18.15.2. Eclipse project

`EclipseProject` has a default instance with the name `eclipseProject`. It generates a `.project` file.

Table 18.17. Java plugin - Eclipse project properties

Task Property	Convention Property
<code>name</code>	<code>project.name</code>
<code>projectType</code>	<code>ProjectType.JAVA</code>

The java plugin also provides a task called `eclipse` which generates both of the eclipse tasks mentioned above. If you are using the war plugin, `eclipse` also leads to the execution of the `eclipseWtp` task.

[15] If any of these properties is empty the trailing `-` is not added to the name.

The Groovy Plugin

The Groovy Plugin extends the Java Plugin to add support for Groovy projects. It can deal with Groovy-only projects and with mixed Java/Groovy projects. It can even deal with Java-only projects. ^[16] The Groovy plugin supports joint compilation of Java and Groovy. This means that your project can contain Groovy classes which use Java classes, and vice versa.

19.1. Tasks

The Groovy plugin adds the following tasks to the project.

Table 19.1. Groovy plugin - tasks

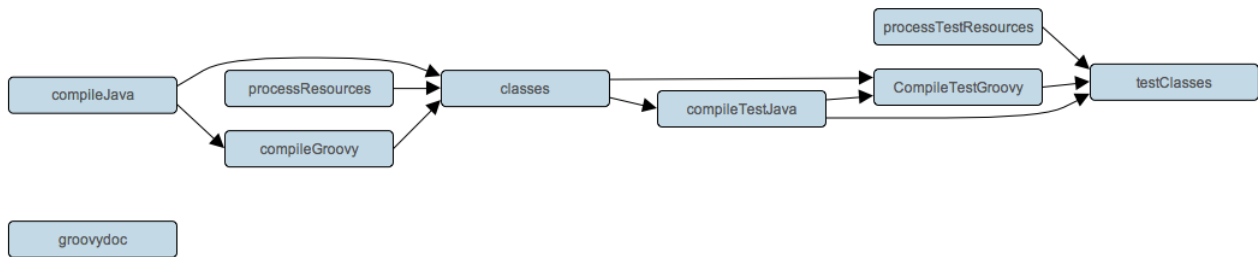
Task name	Depends on	Type	Description
<code>compileGroovy</code>	<code>compileJava</code>	<u>GroovyCompile</u>	Compiles production Groovy source files using groovyc.
<code>compileTestGroovy</code>	<code>compileTestJava</code>	<u>GroovyCompile</u>	Compiles test Groovy source files using groovyc.
<code>compileSourceSetGroovy</code>	<code>compileSourceSetJava</code>	<u>GroovyCompile</u>	Compiles the given source set's Groovy source files using groovyc.
<code>groovydoc</code>	-	<u>Groovydoc</u>	Generates API documentation for the production Groovy source files using groovydoc.

The Groovy plugin adds the following dependencies to tasks added by the Java plugin.

Table 19.2. Groovy plugin - additional task dependencies

Task name	Depends on
<code>classes</code>	<code>compileGroovy</code>
<code>testClasses</code>	<code>compileTestGroovy</code>
<code>sourceSetClasses</code>	<code>compileSourceSetGroovy</code>

Figure 19.1. Groovy plugin - tasks



19.2. Project layout

The Groovy plugin assumes the project layout shown in Table 19.3, “Groovy plugin - project layout”. All the Groovy source directories can contain Groovy *and* Java code. The Java source directories may only contain Java source code. ^[17] None of these directories need exist or have anything in them. The Groovy plugin will compile whatever it finds, and handles anything which is missing.

Table 19.3. Groovy plugin - project layout

Directory	Meaning
src/main/java	Production Java source
src/main/resources	Production resources
src/main/groovy	Production Groovy source. May also contain Java source for joint compilation.
src/test/java	Test Java source
src/test/resources	Test resources
src/test/groovy	Test Groovy source. May also contain Java source for joint compilation.
src/sourceSet/java	Java source for the given source set
src/sourceSet/resources	Resources for the given source set
src/sourceSet/groovy	Groovy source for the given source set. May also contain Java source for joint compilation.

19.2.1. Changing the project layout

TBD

Example 19.1. Custom Groovy source layout

build.gradle

```

sourceSets {
    main {
        groovy {
            srcDir 'src/groovy'
        }
    }
}

```


19.3. Dependency management

The Groovy plugin adds a dependency configuration called `groovy`.

Gradle is written in Groovy and allows you to write your build scripts in Groovy. But this is an internal aspect of Gradle which is strictly separated from building Groovy projects. You are free to choose the Groovy version your project should be build with. This Groovy version is not just used for compiling your code and running your tests. The `groovyc` compiler and the `groovydoc` tool are also taken from the Groovy version you provide. As usual, with freedom comes responsibility ;). You are not just free to choose a Groovy version, you have to provide one. Gradle expects that the groovy libraries are assigned to the `groovy` dependency configuration. Here is an example using the public Maven repository:

Example 19.2. Configuration of Groovy plugin

build.gradle

```
repositories {
    mavenCentral()
}

dependencies {
    groovy group: 'org.codehaus.groovy', name: 'groovy', version: '1.6.0'
}
```

And here is an example using the Groovy JARs checked into the `lib` directory of the source tree:

Example 19.3. Configuration of Groovy plugin

build.gradle

```
repositories {
    flatDir(dirs: file('lib'))
}

dependencies {
    groovy module(':groovy:1.6.0') {
        dependency('asm:asm-all:2.2.3')
        dependency('antlr:antlr:2.7.7')
        dependency('commons-cli:commons-cli:1.2')
        module(':ant:1.7.0') {
            dependencies('ant-junit:1.7.0:jar', 'ant-launcher:1.7.0')
        }
    }
}
```

19.4. Convention properties

The Groovy plugin does not add any convention properties to the project.

19.5. Source set properties

The Groovy plugin adds the following convention properties to each source set in the project. You can use these properties in your build script as though they were properties of the source set object (see [Section 17.2, “Using the convention object”](#)).

Table 19.4. Groovy plugin - source set properties

Property name	Type	Default value	Description
groovy	SourceDirectorySet (read-only)	Not null	The Groovy source files of this source set. Contains all <code>.groovy</code> and <code>.java</code> files found in the Groovy source directories, and excludes all other types of files.
groovy.srcDirs	Set<File> . Can set using anything described in Section 14.4, “Specifying a set of files” .	<code>[projectDir/src/main/groovy]</code>	The source directories containing the Groovy source files of this source set. May also contain Java source files for joint compilation.
allGroovy	FileTree (read-only)	Not null	All Groovy source files of this source set. Contains only the <code>.groovy</code> files found in the Groovy source directories.

These properties are provided by a convention object of type [GroovySourceSet](#).

The Groovy plugin also modifies some source set properties:

Table 19.5. Groovy plugin - source set properties

Property name	Change
allJava	Adds all <code>.java</code> files found in the Groovy source directories.
allSource	Adds all source files found in the Groovy source directories.

19.6. CompileGroovy

The Groovy plugin adds a [GroovyCompile](#) instance for each source set in the project. The task type extends the [Compile](#) task (see [Section 18.10, “CompileJava”](#)). The compile task delegates to the Ant Groovyc task to do the compile. Via the compile task you can set most of the properties of Ants Groovyc task.

Table 19.6. Groovy plugin - CompileGroovy properties

Task Property	Type	Default Value
classpath	FileCollection	<code>sourceSet.compileClasspath</code>
source	FileTree . Can set using anything described in Section 14.4, “Specifying a set of files” .	<code>sourceSet.groovy</code>
destinationDir	File .	<code>sourceSet.classesDir</code>
groovyClasspath	FileCollection	groovy configuration

[16] We don't recommend this, as the Groovy plugin uses the *Groovyc* Ant task to compile the sources. For pure Java projects you might rather stick with `javac`. In particular as you would have to supply a groovy jar for doing this.

[17] We are using the same conventions as introduced by Russel Winders Gant tool (<http://gant.codehaus.org>).

The Scala Plugin

The Scala Plugin extends the Java Plugin to add support for Scala projects. It can deal with Scala-only projects and with mixed Java/Scala projects. It can even deal with Java-only projects. The Scala plugin supports joint compilation of Java and Scala source. This means your project can contain Scala classes which use Java classes, and vice versa.

20.1. Tasks

The Scala plugin adds the following tasks to the project.

Table 20.1. Scala plugin - tasks

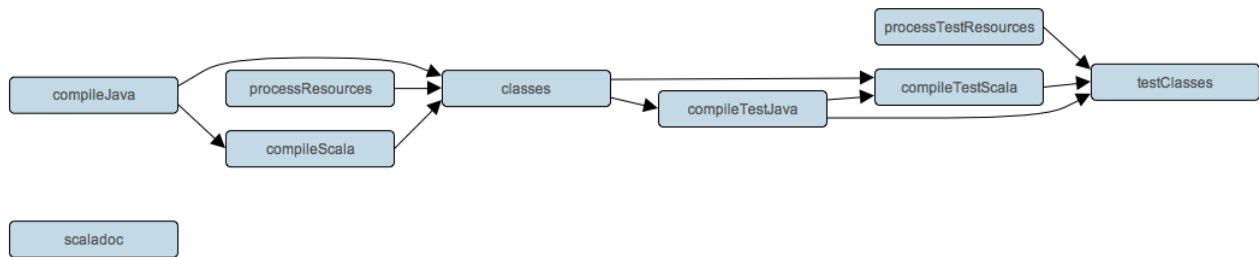
Task name	Depends on	Type	Description
<code>compileScala</code>	<code>compileJava</code>	<u><code>ScalaCompile</code></u>	Compiles production Scala source files using scalac.
<code>compileTestScala</code>	<code>compileTestJava</code>	<u><code>ScalaCompile</code></u>	Compiles test Scala source files using scalac.
<code>compileSourceSetScala</code>	<code>compileSourceSetJava</code>	<u><code>ScalaCompile</code></u>	Compiles the given source set's Scala source files using scalac.
<code>scaladoc</code>	-	<u><code>ScalaDoc</code></u>	Generates API documentation for the production Scala source files using scaladoc.

The Scala plugin adds the following dependencies to tasks added by the Java plugin.

Table 20.2. Scala plugin - additional task dependencies

Task name	Depends on
<code>classes</code>	<code>compileScala</code>
<code>testClasses</code>	<code>compileTestScala</code>
<code>sourceSetClasses</code>	<code>compileSourceSetScala</code>

Figure 20.1. Scala plugin - tasks



20.2. Project layout

The Scala plugin assumes the project layout shown below. All the Scala source directories can contain Scala *and* Java code. The Java source directories may only contain Java source code. None of these directories need exist or have anything in them. The Scala plugin will compile whatever it finds, and handles anything which is missing.

Table 20.3. Scala plugin - project layout

Directory	Meaning
src/main/java	Production Java source
src/main/resources	Production resources
src/main/scala	Production Scala source. May also contain Java source for joint compilation.
src/test/java	Test Java source
src/test/resources	Test resources
src/test/scala	Test Scala source. May also contain Java source for joint compilation.
src/sourceSet/java	Java source for the given source set
src/sourceSet/resources	Resources for the given source set
src/sourceSet/scala	Scala source for the given source set. May also contain Java source for joint compilation.

20.2.1. Changing the project layout

TBD

Example 20.1. Custom Scala source layout

build.gradle

```

sourceSets {
    main {
        scala {
            srcDir 'src/scala'
        }
    }
}

```

20.3. Dependency Management

The Scala plugin adds a `scalaTools` configuration, which it uses to locate the Scala tools, such as `scalac`, to use. You must specify the version of Scala to use. Below is an example.

Example 20.2. Declaring the Scala version to use

`build.gradle`

```
usePlugin 'scala'

repositories {
    mavenRepo urls: 'http://scala-tools.org/repo-releases/'
}

dependencies {
    // Libraries needed to run the scala tools
    scalaTools 'org.scala-lang:scala-compiler:2.7.6'
    scalaTools 'org.scala-lang:scala-library:2.7.6'

    // Libraries needed for scala api
    compile 'org.scala-lang:scala-library:2.7.6'
}
```

20.4. Convention Properties

The Scala plugin does not add any convention properties to the project.

20.5. Source set properties

The Scala plugin adds the following convention properties to each source set in the project. You can use these properties in your build script as though they were properties of the source set object (see [Section 17.2, “Using the convention object”](#)).

Table 20.4. Scala plugin - source set properties

Property name	Type	Default value	Description
scala	<u>SourceDirectorySet</u> (read-only)	Not null	The Scala source files of this source set. Contains all .scala and .java files found in the Scala source directories, and excludes all other types of files.
scala.srcDirs	Set<File>. Can set using anything described in <u>Section 14.4, “Specifying a set of files”</u> .	[<i>projectDir</i> /src/main/scala]	The source directories containing the Scala source files of this source set. May also contain Java source files for joint compilation.
allScala	<u>FileTree</u> (read-only)	Not null	All Scala source files of this source set. Contains only the .scala files found in the Scala source directories.

These convention properties are provided by a convention object of type ScalaSourceSet.

The Scala plugin also modifies some source set properties:

Table 20.5. Scala plugin - source set properties

Property name	Change
allJava	Adds all .java files found in the Scala source directories.
allSource	Adds all source files found in the Scala source directories.

21

The War Plugin

The War plugin extends the Java Plugin to add support for assembling web application WAR files. It disables the default JAR archive generation of the Java Plugin and adds a default WAR archive task.

21.1. Tasks

The War plugin adds the following tasks to the project.

Table 21.1. War plugin - tasks

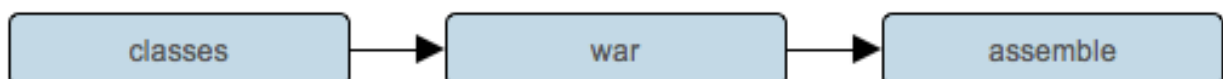
Task name	Depends on	Type	Description
war	compile	War	Assembles the application WAR file.

The War plugin adds the following dependencies to tasks added by the Java plugin.

Table 21.2. War plugin - additional task dependencies

Task name	Depends on
assemble	war

Figure 21.1. War plugin - tasks



21.2. Project layout

Table 21.3. War plugin - project layout

Directory	Meaning
src/main/webapp	Web application sources

21.3. Dependency management

The War plugin adds two dependency configurations: `providedCompile` and `providedRuntime`. Those configurations have the same scope as the respective `compile` and `runtime` configurations, except that they are not added to the WAR archive. It is important to note that those `provided` configurations work transitively. Let's say you add `commons-httpclient:commons-httpclient:3.0` to any of the `provided` configurations. This dependency has a dependency on `commons-codec`. This means neither `httpClient` nor `commons-codec` is added to your WAR, even if `commons-codec` were an explicit dependency of your `compile` configuration. If you don't want this transitive behavior, simply declare your `provided` dependencies like `provided`.

21.4. Convention properties

Table 21.4. War plugin - directory properties

Property name	Type	Default value	Description
<code>webAppDirName</code>	String	<code>src/main/webapp</code>	The name of the web application source directory, relative to the project directory.
<code>webAppDir</code>	File (read-only)	<code>projectDir/webAppDirName</code>	The web application source directory.

These properties are provided by a `WarPluginConvention` convention object.

21.5. War

The default behavior of the War task is to copy the content of `src/main/webapp` to the root of the archive. Your `webapp` folder may of course contain a `WEB-INF` sub-directory, which again may contain a `web.xml` file. Your compiled classes are compiled to `WEB-INF/classes`. All the dependencies of the `runtime` configuration are copied to `WEB-INF/lib`.

Have also a look at `War`.

21.6. Customizing

Here is an example with the most important customization options:

Example 21.1. Customization of war plugin

build.gradle

```
import org.apache.commons.httpclient.HttpClient
import org.apache.commons.httpclient.methods.GetMethod

group = 'gradle'
version = '1.0'
usePlugin('war')
usePlugin('jetty')

configurations {
    moreLibs
}

repositories {
    flatDir(dirs: "$rootDir/lib")
    mavenCentral()
}

dependencies {
    compile ":compile:1.0"
    providedCompile ":providedCompile:1.0@jar", "javax.servlet:servlet-api:2.5"
    runtime ":runtime:1.0"
    providedRuntime ":providedRuntime:1.0@jar"
    testCompile "junit:junit:3.8.2"
    moreLibs ":otherLib:1.0"
}

war {
    fileSet(dir: file('src/rootContent')) // adds a file-set to the root of the archive
    webInf(dir: file('src/additionalWebInf')) // adds a file-set to the WEB-INF dir.
    additionalLibs(dir: file('additionalLibs')) // adds a file-set to the WEB-INF/lib dir.
    libConfigurations('moreLibs') // adds a configuration to the WEB-INF/lib dir.
    webXml = file('src/someWeb.xml') // copies a file to WEB-INF/web.xml
}

jar.enabled = true

[jettyRun, jettyRunWar]*.daemon = true
stopKey = 'foo'
stopPort = 9451
httpPort = 8163

task runTest(dependsOn: jettyRun) << {
    callServlet()
}

task runWarTest(dependsOn: jettyRunWar) << {
    callServlet()
}

private void callServlet() {
    HttpClient client = new HttpClient()
    GetMethod method = new GetMethod("http://localhost:$httpPort/customised/hello")
    client.executeMethod(method)
    new File(buildDir, "servlet-out.txt").write(method.getResponseBodyAsString())
    jettyStop.execute()
}
```

Of course one can configure the different file-sets with a closure to define excludes and includes.

If you want to enable the generation of the default jar archive additional to the war archive just type:

Example 21.2. Generation of JAR archive in addition to WAR archive

build.gradle

```
jar.enabled = true
```

21.7. Eclipse WTP

`EclipseWtp` has a default instance with the name `eclipseWtp`. It generates a `.settings/org.eclipse` file.

[18] The runtime configuration extends the compile configuration.

The Jetty Plugin

The Jetty Plugin extends the War Plugin to add tasks which allow you to deploy your web application to a Jetty web container embedded in the build.

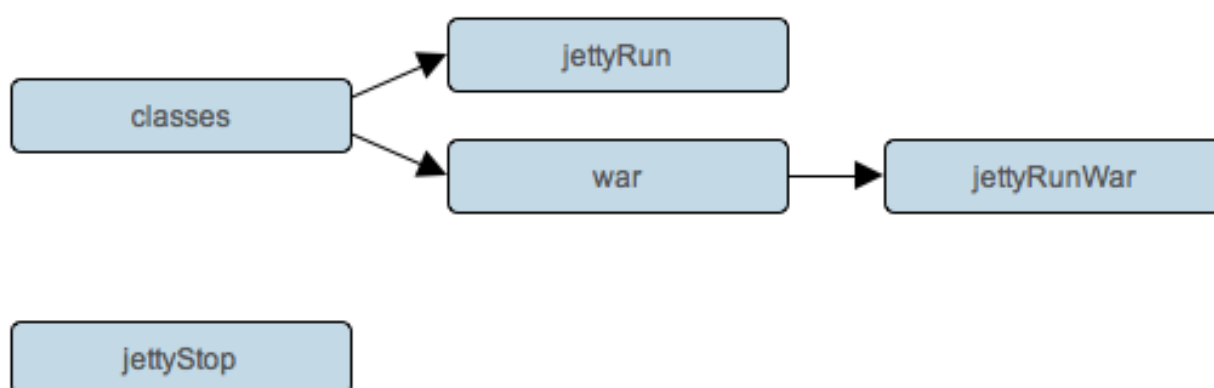
22.1. Tasks

The Jetty plugin defines the following tasks:

Table 22.1. Jetty plugin - tasks

Task name	Depends on	Type	Description
jettyRun	compile	JettyRun	Starts a Jetty instance and deploys the exploded web application to it.
jettyRunWar	war	JettyRunWar	Starts a Jetty instance and deploys the WAR to it.
jettyStop	-	JettyStop	Stops the Jetty instance.

Figure 22.1. Jetty plugin - tasks



22.2. Project layout

The Jetty plugin uses the same layout as the War plugin.

22.3. Dependency management

The Jetty plugin does not define any dependency configurations.

22.4. Convention properties

The Jetty plugin defines the following convention properties:

Table 22.2. Jetty plugin - properties

Property name	Type	Default value	Description
httpPort	Integer	8080	The TCP port which Jetty should listen for HTTP requests on.
stopPort	Integer	null	The TCP port which Jetty should listen for admin requests on.
stopKey	String	null	The key to pass to Jetty when requesting it to stop.

These properties are provided by a JettyPluginConvention convention object.

23

The Maven Plugin

This chapter is a work in progress

23.1. Tasks

TBD

23.2. Project layout

TBD

23.3. Dependency management

TBD

23.4. Convention properties

TBD

The Code Quality Plugin

The code quality plugin adds tasks which perform code quality checks and generate reports from these checks. The following tools are supported:

- [Checkstyle](#)
- [CodeNarc](#)

To use the plugin, add the following to your build script:

```
usePlugin 'code-quality'
```

24.1. Tasks

When used with the Java plugin, the code quality plugin adds the following tasks to the project:

Table 24.1. Code quality plugin - Java tasks

Task name	Depends on	Type	Description
<code>checkstyleMain</code>	-	Checkstyle	Runs Checkstyle against the production Java source files.
<code>checkstyleTest</code>	<code>compile</code>	Checkstyle	Runs Checkstyle against the test Java source files.
<code>checkstyleSourceSet</code>		Checkstyle	Runs Checkstyle against the given source set's Java source files.

When used with the Groovy plugin, the code quality plugin adds the following tasks to the project:

Table 24.2. Code quality plugin - tasks

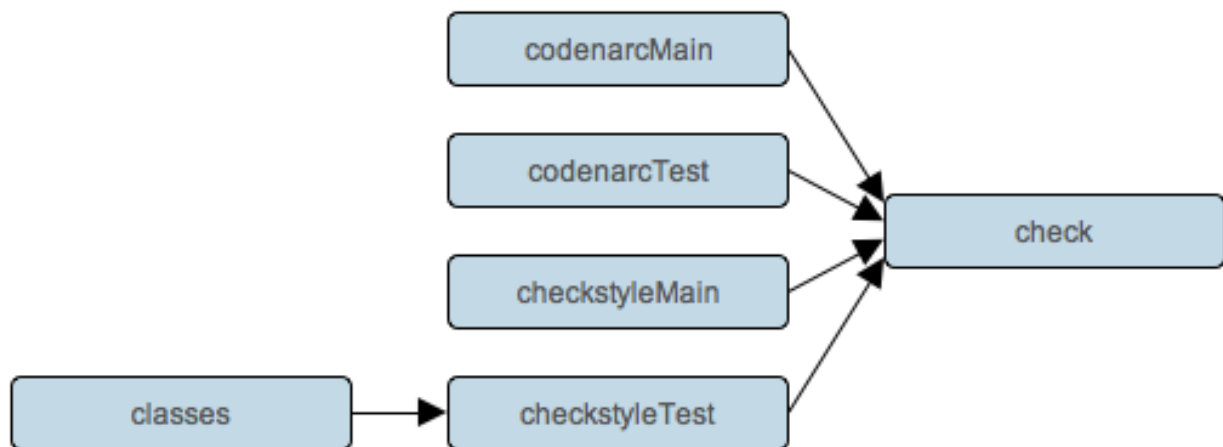
Task name	Depends on	Type	Description
codenarcMain	-	<u>CodeNarc</u>	Runs CodeNarc against the production Groovy source files.
codenarcTest	-	<u>CodeNarc</u>	Runs CodeNarc against the test Groovy source files.
codenarc <i>SourceSet</i>	-	<u>CodeNarc</u>	Runs CodeNarc against the given source set's Groovy source files.

The Code quality plugin adds the following dependencies to tasks added by the Java plugin.

Table 24.3. Code quality plugin - additional task dependencies

Task name	Depends on
check	All Checkstyle and CodeNarc tasks, including checkstyleMain, checkstyleTest, codenarcMain and codenarcTest

Figure 24.1. Code quality plugin - tasks



24.2. Project layout

The code quality plugin expects the following project layout:

Table 24.4. Code quality plugin - project layout

File	Meaning
config/checkstyle/checkstyle.xml	Checkstyle configuration file
config/codenarc/codenarc.xml	CodeNarc configuration file

24.3. Dependency management

The code quality plugin does not add any dependency configurations.

24.4. Convention properties

When used with the Java plugin, the code quality plugin adds the following convention properties to the project:

Table 24.5. Code quality plugin - convention properties

Property name	Type	Default value	Description
<code>checkstyleConfigFileName</code>	String	<code>config/checkstyle/checkstyle.xml</code>	The location of the Checkstyle configuration file, relative to the project directory.
<code>checkstyleConfigFile</code>	File (read-only)	<code>projectDir/checkstyleConfigFileName</code>	The Checkstyle configuration file.
<code>checkstyleResultsDirName</code>	String	<code>checkstyle</code>	The name of the directory to generate Checkstyle results in, relative to the build directory.
<code>checkstyleResultsDir</code>	File (read-only)	<code>buildDir/checkstyleResultsDirName</code>	The directory to generate Checkstyle results in.

These convention properties are provided by a convention object of type `JavaCodeQualityPluginConvention`.

When used with the Groovy plugin, the code quality plugin adds the following convention properties to the project:

Table 24.6. Code quality plugin - convention properties

Property name	Type	Default value	Description
<code>codeNarcConfigFileName</code>	String	<code>config/codenarc/codenarc.xml</code>	The location of the CodeNarc configuration file, relative to the project directory.
<code>codeNarcConfigFile</code>	File (read-only)	<code>projectDir/codeNarcConfigFileName</code>	The CodeNarc configuration file.
<code>codeNarcReportsDirName</code>	String	<code>codenarc</code>	The name of the directory to generate CodeNarc reports into, relative to the reports directory.
<code>codeNarcReportsDir</code>	File (read-only)	<code>reportsDir/codeNarcReportsDirName</code>	The directory to generate CodeNarc reports into.

These convention properties are provided by a convention object of type `GroovyCodeQualityPluginConvention`.

The OSGi Plugin

The Gradle OSGi plugin enables the generation of an OSGi manifest. This OSGi manifest is automatically added to all the JAR files produced by the project. This plugin makes heavy use of Peter Kriens [BND tool](#).

25.1. Tasks

TBD

25.2. Project layout

TBD

25.3. Dependency management

TBD

25.4. Convention properties

The OSGi plugin adds an `osgi` property to every `jar` task. This `osgi` property points to an instance of [OsgiManifest](#). Via the `OsgiManifest` object you can control the generation of the OSGi Manifest of the respective jar. The OSGi plugin assign default values to the `OsgiManifest` object.

Table 25.1. OSGi properties

Task Property	Convention Property
classesDir	project.classesDir
version	project.version
name	project.archivesBaseName
symbolicName	transformation of the name and the group to produce a valid OSGi symbolic name
classpath	project.dependencies.resolve('runtime')

The classes in the classes dir are analyzed regarding there package dependencies and the packages they expose. Based on this the *Import-Package* and the *Export-Package* values of the OSGi Manifest are

calculated. If the classpath contains jars with an OSGi bundle, the bundle information is used to specify version information for the *Import-Package* value. Beside the explicit properties of the `OsgiManifest` object you can add instructions.

Example 25.1. Configuration of OSGi MANIFEST.MF file

build.gradle

```
configure(jar.osgi) {
    name = 'overwrittenSpecialOsgiName'
    instruction 'Private-Package',
        'org.mycomp.package1',
        'org.mycomp.package2'
    instruction 'Bundle-Vendor', 'MyCompany'
    instruction 'Bundle-Description', 'Platform2: Metrics 2 Measures Framework'
    instruction 'Bundle-DocURL', 'http://www.mycompany.com'
}
```

The first argument of the instruction call is the key of the property. The other arguments form the value. They are joined by Gradle with the `,` separator. To learn more about the available instructions have a look at the [BND tool](#).

The Eclipse Plugin

This chapter is a work in progress.

The Eclipse plugin generates files that are used by Eclipse IDE, thus making it possible to import the project into Eclipse (File - Import... - Existing Projects into Workspace). Both external and project dependencies are considered.

To use the plugin, include in your build script:

```
usePlugin('eclipse')
```

The Eclipse plugin will create different files depending on the other plugins used. If used together with the Java Plugin, `.project` and `.classpath` files will be generated. If used with the War plugin, additional `wtm` files will be generated.

26.1. Usage

There are several tasks (presented in Table 26.1, "Eclipse plugin - tasks") that the Eclipse plugin provides, but you will probably use only `eclipse` task by executing `gradle eclipse`.

26.2. Tasks

The Eclipse plugin adds the tasks shown in Table 26.1, "Eclipse plugin - tasks" to a project.

Table 26.1. Eclipse plugin - tasks

Task name	Depends on	Type
<code>eclipse</code>	<code>eclipseCp</code> , <code>eclipseProject</code> , <code>eclipseWtp</code>	-
<code>eclipseClean</code>	-	<u>EclipseClean</u>
<code>eclipseCp</code>	-	<u>EclipseClasspath</u>
<code>eclipseProject</code>	-	<u>EclipseProject</u>
<code>eclipseWtp</code>	-	<u>EclipseWtp</u>
<code>eclipseWtpModule</code>	-	<u>EclipseWtpModule</u>

The Project Report Plugin

The Project report plugin is currently a work in progress, and at this stage doesn't do particularly much. We plan to add much more to these reports in the next release of Gradle.

The Project report plugin adds some tasks to your project which generate reports containing useful information about your build.

27.1. Tasks

The project report plugin defines the following tasks:

Table 27.1. Project report plugin - tasks

Task name	Depends on	Type	Description
dependencyReport	-	<u>DependencyReportTask</u>	Generates the project dependency report.
propertyReport	-	<u>PropertyReportTask</u>	Generates the project property report.
taskReport	-	<u>TaskReportTask</u>	Generates the project task report.
projectReport	dependencyReport, propertyReport, taskReport	<u>Task</u>	Generates all project reports.

27.2. Project layout

The project report plugin does not require any particular project layout.

27.3. Dependency management

The project report plugin does not define any dependency configurations.

27.4. Convention properties

The project report defines the following convention properties:

Table 27.2. Project report plugin - directory properties

Property name	Type	Default value	Description
reportsDirName	String	reports	The name of the directory to generate reports into, relative to the build directory.
reportsDir	File (read-only)	<i>buildDir/reportsDirName</i>	The directory to generate reports into.
projectReportDirName	String	project	The name of the directory to generate the project report into, relative to the reports directory.
projectReportDir	File (read-only)	<i>reportsDir/projectReportDirName</i>	The directory to generate the project report into.

These convention properties are provided by a convention object of type `ProjectReportsPluginConvention`.

Dependency Management

28.1. Introduction

This chapter gives an overview of issues related with dependency management and presents how Gradle can be used to overcome them.

Gradle offers a very good support for dependency management. If you are familiar with Maven or Ivy approach you will be delighted to learn that:

- All the concepts that you already know and like are still there and are fully supported by Gradle. The current dependency management solutions all require to work with XML descriptor files and are usually based on remote repositories for downloading the dependencies. Gradle fully supports this approach.
- Gradle works *perfectly* with your existent dependency management infrastructure, be it Maven or Ivy. All the repositories you have set up with your custom pom or ivy files can be used as they are. No changes necessary.
- Additionally, Gradle offers a simpler approach, which might be better suited for some projects.

28.2. Dependency management overview

We think dependency management is very important for almost any project. Yet the kind of dependency management you need depends on the complexity and the environment of your project. Is your project a distribution or a library? Is it part of an enterprise environment, where it is integrated into other projects builds or not? But all types of projects share the following requirements:

- The version of the jar must be easy to recognize. Sometimes the version is in the Manifest file of the jar, often not. And even if, it is rather painful to always look into the Manifest file to learn about the version. Therefore we think that you should only use jars which have their version as part of their file name.
- It hopes to be clear what are the first level dependencies and what are the transitive ones. There are different ways to achieve this. We will look at this later.
- Conflicting versions of the same jar should be detected and either resolved or cause an exception.

28.2.1. Versioning the jar name

Why do we think this is necessary? Without a dependency management as described above, you are likely to burn your fingers sooner or later. If it is unclear which version of a jar you are using, this can introduce subtle bugs which are very hard to find. For example there might be a project which uses Hibernate 3.0.4. There are some problems with Hibernate so a developer installs version 3.0.5 of Hibernate on her machine. This did not solve the problem but she forgot to roll back Hibernate to 3.0.4. Weeks later there is an exception on the integration machine which can't be reproduced on the developer machine. Without a version in the jar name this problem might take a long time to debug. Version in the jar names increases the expressiveness of your project and makes it easier to maintain.

28.2.2. Transitive dependency management

Why is transitive dependency management so important? If you don't know which dependencies are first level dependencies and which ones are transitive you will soon lose control over your build. Even Gradle has already 20+ jars. An enterprise project using Spring, Hibernate, etc. easily ends up with 100+ jars. There is no way to memorize where all these jars come from. If you want to get rid of a first level dependency you can't be sure which other jars you should remove. Because a dependency of a first level dependency might also be a first level dependency itself. Or it might be a transitive dependency of another of your first level dependencies. Many first level dependencies are runtime dependencies and the transitive dependencies are of course all runtime dependencies. So the compiler won't help you much here. The end of the story is, as we have seen very often, no one dares to remove any jar any longer. The project classpath is a complete mess and if a classpath problem arises, hell on earth invites you for a ride. In one of my former projects, I found some ldap related jar in the classpath, whose sheer presence, as I found out after much research, accelerated LDAP access. So removing this jar would not have led to any errors at compile or runtime.

Gradle offers you different ways to express what are first level and what are transitive dependencies. Gradle allows you for example to store your jars in CVS or SVN without XML descriptor files and still use transitive dependency management. Gradle also validates your dependency hierarchy against the reality of your code by using only the first level dependencies for compiling.

28.2.3. Version conflicts

In your dependency description you tell Gradle which version of a dependency is needed by another dependency. This frequently leads to conflicts. Different dependencies rely on different versions of another dependency. The JVM unfortunately does not offer yet any easy way, to have different versions of the same jar in the classpath (see [Section 28.2.4, "Dependency management and Java"](#)). What Gradle offers you is a resolution strategy, by default the newest version is used. To deal with problems due to version conflicts, reports with dependency graphs are also very helpful. Such reports are another feature of dependency management.

28.2.4. Dependency management and Java

Traditionally, Java has offered no support at all for dealing with libraries and versions. There are no standard ways to say that `foo-1.0.jar` depends on a `bar-2.0.jar`. This has led to proprietary solutions. The most popular ones are Maven and Ivy. Maven is a complete build system whereas Ivy focuses solely on dependency management.

Both approaches rely on descriptor XML files, which contains information about the dependencies of a particular jar. Both also use repositories where the actual jars are placed together with their descriptor files. And both offer resolution for conflicting jar versions in one form or the other. Yet we think the differences of

both approaches are significant in terms of flexibility and maintainability. Beside this, Ivy fully supports the Maven dependency handling. So with Ivy you have access to both worlds. We like Ivy very much. Gradle uses it under the hood for its dependency management. Ivy is most often used via Ant and XML descriptors. But it also has an API. We integrate deeply with Ivy via its API. This enables us to build new concepts on top of Ivy which Ivy does not offer itself.

Right now there is a lot of movement in the field of dependency handling. There is OSGi and there is JSR-294. ^[19] OSGi is available already, JSR-294 is supposed to be shipped with Java 7. These technologies deal, amongst many other things, also with a painful problem which is neither solved by Maven nor by Ivy. This is enabling different versions of the same jar to be used at runtime.

28.3. How to declare your dependencies

People who know Ivy have come across most of the concepts we are going to introduce now. But Gradle does not use any XML for declaring the dependencies (e.g. no `ivy.xml` file). It has its own notation which is part of the Gradle build file.

28.3.1. Configurations

Dependencies are grouped in configurations. Configurations have a name, a number of other properties, and they can extend each other. For examples see: [Section 8.1, “Artifact configurations”](#). If you use the Java plugin, Gradle adds a number of pre-defined configurations to your build. The plugin also assigns configurations to tasks. See [Section 18.4, “Dependency management”](#) for details. Of course you can add your add custom configurations on top of that. There are many use cases for custom configurations. This is very handy for example for adding dependencies not needed for building or testing your software (e.g. additional JDBC drivers to be shipped with your distribution). The configurations are managed by a `configu` object. The closure you pass to the configurations object is applied against its API. To learn more about this API have a look at [ConfigurationHandler](#).

28.3.2. Module dependencies

Module dependencies are the most common dependencies. They correspond to a dependency in an external repository.

Example 28.1. Module dependencies

build.gradle

```
dependencies {
    runtime group: 'org.springframework', name: 'spring-core', version: '2.5'
    runtime 'org.springframework:spring-core:2.5', 'org.springframework:spring-aop:2.5'
    runtime(
        [group: 'org.springframework', name: 'spring-core', version: '2.5'],
        [group: 'org.springframework', name: 'spring-aop', version: '2.5']
    )
    runtime('org.hibernate:hibernate:3.0.5') {
        transitive = true
    }
    runtime group: 'org.hibernate', name: 'hibernate', version: '3.0.5', transitive: true
    runtime(group: 'org.hibernate', name: 'hibernate', version: '3.0.5') {
        transitive = true
    }
}
```

Gradle provides different notations for module dependencies. There is a string notation and a map notation.

A module dependency has an API which allows for further configuration. Have a look at [ExternalModuleDependency](#) to learn all about the API. This API provides properties and configuration methods. Via the string notation you can define a subset the properties. With the map notation you can define all properties. To have access to the complete API, either with the map or with the string notation, you can assign a single dependency to a configuration together with a closure.

If you declare a module dependency, Gradle looks for a corresponding module descriptor file (`pom.xml` or `ivy.xml`) in the repositories. If such a module descriptor file exists, it is parsed and the artifacts of this module (e.g. `hibernate-core-3.0.5.jar`) as well as its dependencies (e.g. `cglib`) are downloaded. If no such module descriptor file exists, Gradle looks for a file called `hibernate-3.0.5.jar` to retrieve. In Maven a module can only have one and only one artifact. In Gradle and Ivy a module can have multiple artifacts. Each artifact can have a different set of dependencies.

28.3.2.1. Artifact only notation

As said above, if no module descriptor file can be found, Gradle by default downloads a jar with the name of the module. But sometimes, even if the repository contains module descriptors, you want to download only the artifact jar, without the dependencies. ^[20] And sometimes you want to download a zip from a repository, that does not have module descriptors. Gradle provides an *artifact only* notation for those use cases - simply prefix the extension that you want to be downloaded with '@' sign:

Example 28.2. Artifact only notation

build.gradle

```
dependencies {
    runtime "org.groovy:groovy:1.5.6@jar"
    runtime group: 'org.groovy', name: 'groovy', version: '1.5.6', ext: 'jar'
}
```

An artifact only notation creates a module dependency which downloads only the artifact file with the specified extension. Existing module descriptors are ignored.

28.3.2.2. Classifiers

The Maven dependency management has the notion of classifiers. ^[21] Gradle supports this. To retrieve classified dependencies from a maven repository you can write:

Example 28.3. Dependency with classifier

build.gradle

```
compile "org.gradle.test.classifiers:service:1.0:jdk15@jar"
otherConf group: 'org.gradle.test.classifiers', name: 'service', version: '1.0', classifier: 'jdk15'
```

As you can see in the example, classifiers can be used together with setting an explicit extension (artifact only notation).

28.3.3. Client module dependencies

Client module dependencies enable you to declare *transitive* dependencies directly in your build script. They are a replacement for a module descriptor XML file in an external repository.

Example 28.4. Client module dependencies - transitive dependencies

build.gradle

```
dependencies {
    runtime module("org.codehaus.groovy:groovy-all:1.5.6") {
        dependency("commons-cli:commons-cli:1.0") {
            transitive = false
        }
        module(group: 'org.apache.ant', name: 'ant', version: '1.7.0') {
            dependencies "org.apache.ant:ant-launcher:1.7.0@jar", "org.apache.ant:ant-:
        }
    }
}
```

This declares a dependency of your project on Groovy. Groovy itself has dependencies. But Gradle does not look for an XML descriptor to figure them out but gets the information from the build file. The dependencies of a client module can be normal module dependencies or artifact dependencies or another client module. Have also a look at the javadoc: [ClientModule](#)

In the current release client modules have one limitation. Let's say your project is a library and you want this library to be uploaded to your company's Maven or Ivy repository. Gradle uploads the jars of your project to the company repository together with the XML descriptor file of the dependencies. If you use client modules the dependency declaration in the XML descriptor file is not correct. We will improve this in a future release of Gradle.

28.3.4. Project dependencies

Gradle distinguishes between external dependencies and dependencies on projects which are part of the same multi-project build. For the latter you can declare *Project Dependencies*.

Example 28.5. Project dependencies

build.gradle

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

For more information see the javadoc for [ProjectDependency](#)

Multi-project builds are discussed in [Chapter 31, Multi-project Builds](#).

Transitive compile dependencies are disabled by default. They can be enabled using:

```
configurations.compile.transitive = true
```

28.3.5. File dependencies

File dependencies allow you to directly add a set of files to a configuration, without first adding them to a repository. This can be useful if you cannot, or do not want to, place certain files in a repository. Or if you do not want to use any repositories at all for storing your dependencies.

To add some files as a dependency for a configuration, you simply pass a [file collection](#) as a dependency:

Example 28.6. File dependencies

build.gradle

```
dependencies {
    runtime files('libs/a.jar', 'libs/b.jar')
    runtime fileTree(dir: 'libs', includes: ['*.jar'])
}
```

File dependencies are not included in the published dependency descriptor for your project. However, file dependencies are included in transitive project dependencies within the same build. This means they cannot be used outside the current build, but they can be used with the same build.

You can declare which tasks produce the files for a file dependency. You might do this when, for example, the files are generated by the build.

Example 28.7. Generated file dependencies

build.gradle

```
dependencies {
    compile files("$buildDir/classes") {
        builtBy 'compile'
    }
}

task compile << {
    println 'compiling classes'
}

task list(dependsOn: configurations.compile) << {
    println "classpath = ${configurations.compile.collect {File file -> file.name}}"
}
```

Output of **gradle -q list**

```
> gradle -q list
compiling classes
classpath = [classes]
```

28.3.6. Excluding transitive dependencies

You can exclude a *transitive* dependency either by configuration or by dependency:

Example 28.8. Excluding transitive dependencies

build.gradle

```
configurations {
    compile.exclude module: 'commons'
    all*.exclude group: 'org.gradle.test.excludes', module: 'reports'
}

dependencies {
    compile("org.gradle.test.excludes:api:1.0") {
        exclude module: 'shared'
    }
}
```

If you define an exclude for a particular configuration, the excluded transitive dependency will be filtered for all dependencies when resolving this configuration or any inheriting configuration. If you want to exclude a transitive dependency from all your configurations you can use the Groovy spread-dot operator to express this in a concise way, as shown in the example. When defining an exclude, you can specify either only the organization or only the module name or both. Have also a look at the javadoc of [Dependency](#) and [Configuration](#).

28.3.7. Optional attributes

All attributes for a dependency are optional, except the name. It depends on the repository type, which information is need for actually finding the dependencies in the repository. See [Section 28.5, “Repositories”](#). If you work for example with Maven repositories, you need to define the group, name and version. If you work with filesystem repositories you might only need the name or the name and the version.

Example 28.9. Optional attributes of dependencies

build.gradle

```
dependencies {
    runtime ":junit:4.4", ":testng"
    runtime name: 'testng'
}
```

You can also assign collections or arrays of dependency notations to a configuration:

Example 28.10. Collections and arrays of dependencies

build.gradle

```
List groovy = ["org.codehaus.groovy:groovy-all:1.5.4@jar",
               "commons-cli:commons-cli:1.0@jar",
               "org.apache.ant:ant:1.7.0@jar"]
List hibernate = ['org.hibernate:hibernate:3.0.5@jar', 'somegroup:someorg:1.0@jar']
dependencies {
    runtime groovy, hibernate
}
```

28.3.8. Dependency configurations

In Gradle a dependency can have different configurations (as your project can have different configurations). If you don't specify anything explicitly, Gradle uses the default configuration of the dependency. For dependencies from a Maven repository, the default configuration is the only available one anyway. If you work with Ivy repositories and want to declare a non-default configuration for your dependency you have to use the map notation and declare:

Example 28.11. Dependency configurations

build.gradle

```
dependencies {
    runtime group: 'org.somegroup', name: 'somedependency', version: '1.0', configurat:
}
```

To do the same for project dependencies you need to declare:

Example 28.12. Dependency configurations for project

build.gradle

```
dependencies {  
    compile project(path: ':api', configuration: 'spi')  
}
```

28.3.9. Dependency reports

You can generate dependency reports from the command line (see [Section 9.5, “Obtaining information about your build”](#)). With the help of the Project report plugin (see [Chapter 27, *The Project Report Plugin*](#)) such a report can be created by your build.

28.4. Working with dependencies

For the examples below we have the following dependencies setup:

Example 28.13. Configuration.copy

build.gradle

```
configurations {  
    sealife  
    alllife.extendsFrom sealife  
}  
  
dependencies {  
    sealife "sea.mammals:orca:1.0", "sea.fish:shark:1.0", "sea.fish:tuna:1.0"  
    alllife "air.birds:albatros:1.0"  
}
```

The dependencies have the following transitive dependencies:

shark-1.0 -> seal-2.0, tuna-1.0

orca-1.0 -> seal-1.0

tuna-1.0 -> herring-1.0

You can use the configuration to access the declared dependencies or a subset of those:

Example 28.14. Accessing declared dependencies

build.gradle

```
task dependencies << {
    configurations.alllife.dependencies.each { dep -> println dep.name }
    println()
    configurations.alllife.allDependencies.each { dep -> println dep.name }
    println()
    configurations.alllife.allDependencies.findAll { dep -> dep.name != 'orca' }.each {
    }
}
```

Output of **gradle -q dependencies**

```
> gradle -q dependencies
albatros

albatros
orca
shark
tuna

albatros
shark
tuna
```

`dependencies` returns only the dependencies belonging explicitly to the configuration. `allDependencies` includes the dependencies from extended configurations.

To get the library files of the configuration dependencies you can do:

Example 28.15. Configuration.files

build.gradle

```
task allFiles << {
    configurations.sealife.files.each { file ->
        println file.name
    }
}
```

Output of **gradle -q allFiles**

```
> gradle -q allFiles
orca-1.0.jar
shark-1.0.jar
seal-2.0.jar
tuna-1.0.jar
herring-1.0.jar
```

Sometimes you want the library files of a subset of the configuration dependencies (e.g. of a single dependency).

Example 28.16. Configuration.files with spec

build.gradle

```
task files << {
    configurations.sealife.files { dep -> dep.name == 'orca' }.each { file ->
        println file.name
    }
}
```

Output of **gradle -q files**

```
> gradle -q files
orca-1.0.jar
seal-2.0.jar
```

The `Configuration.files` method always retrieves all artifacts of the *whole* configuration. It then filters the retrieved files by specified dependencies. As you can see in the example, transitive dependencies are included.

You can also copy a configuration. You can optionally specify that only a subset of dependencies from the original configuration should be copied. The copying methods come in two flavors. The `copy` method copies only the dependencies belonging explicitly to the configuration. The `copyRecursive` method copies all the dependencies, including the dependencies from extended configurations.

Example 28.17. Configuration.copy

build.gradle

```
task copy << {
    configurations.alllife.copyRecursive { dep -> dep.name != 'orca' }.allDependencies.
        println dep.name
    }
    println()
    configurations.alllife.copy().allDependencies.each { dep ->
        println dep.name
    }
}
```

Output of **gradle -q copy**

```
> gradle -q copy
albatros
shark
tuna

albatros
```

It is important to note that the returned files of the copied configuration are often but not always the same than the returned files of the dependency subset of the original configuration. In case of version conflicts between dependencies of the subset and dependencies not belonging to the subset the resolve result might be different.

Example 28.18. Configuration.copy vs. Configuration.files

build.gradle

```
task copyVsFiles << {
    configurations.sealife.copyRecursive { dep -> dep.name == 'orca' }.each { file ->
        println file.name
    }
    println()
    configurations.sealife.files { dep -> dep.name == 'orca' }.each { file ->
        println file.name
    }
}
```

Output of `gradle -q copyVsFiles`

```
> gradle -q copyVsFiles
orca-1.0.jar
seal-1.0.jar

orca-1.0.jar
seal-2.0.jar
```

In the example above, `orca` has a dependency on `seal-1.0` whereas `shark` has a dependency on `seal-2.0`. The original configuration has therefore a version conflict which is resolved to the newer `seal-2.0` version. The `files` method therefore returns `seal-2.0` as a transitive dependency of `orca`. The copied configuration only has `orca` as a dependency and therefore there is no version conflict and `seal-1.0` is returned as a transitive dependency.

Once a configuration is resolved it is immutable. Changing its state or the state of one of its dependencies will cause an exception. You can always copy a resolved configuration. The copied configuration is in the unresolved state and can be freshly resolved.

To learn more about the API of the configuration class see the javadoc: [Configuration](#).

28.5. Repositories

28.5.1. Introduction

The Gradle repository management, based on Apache Ivy, gives you have a lot of freedom regarding the repository layout and the retrieval policies. Additionally Gradle provides a couple of convenience method to add preconfigured repositories.

28.5.2. Maven repositories

To add the central Maven2 repository (<http://repo1.maven.org/maven2>) simply type:

Example 28.19. Adding central Maven repository

build.gradle

```
repositories {
    mavenCentral()
}
```

Now Gradle looks for your dependencies in this repository.

Quite often certain jars are not in the official Maven repository for licensing reasons (e.g. JTA), but its poms are.

Example 28.20. Adding many Maven repositories

build.gradle

```
repositories {
    mavenCentral name: 'single-jar-repo', urls: "http://repo.mycompany.com/jars"
    mavenCentral name: 'multi-jar-repos', urls: ["http://repo.mycompany.com/jars1", "ht
```

Gradle looks first in the central Maven repository for the pom and the jar. If the jar can't be found there, its looks for it in the other repositories.

For adding a custom Maven repository you can say:

Example 28.21. Adding custom Maven repository

build.gradle

```
repositories {
    mavenRepo urls: "http://repo.mycompany.com/maven2"
}
```

To declare additional repositories to look for jars (like above in the example for the central Maven repository), you can say:

Example 28.22. Adding additional Maven repositories for JAR files

build.gradle

```
repositories {
    mavenRepo urls: ["http://repo2.mycompany.com/maven2", "http://repo.mycompany.com/ja
```

The first URL is used to look for poms and jars. The subsequent URLs are used to look for jars.

28.5.2.1. Accessing password protected Maven repositories

To access a password protected Maven repository (basic authentication) you need to use one of Ivy features:

Example 28.23. Accessing password protected Maven repository

build.gradle

```
org.apache.ivy.util.url.CredentialsStore.INSTANCE.addCredentials("REALM", "HOST", "USEF
```

Host name should not include "http://" prefix. It is advisable to keep your login and password in gradle. rather than directly in the build file.

28.5.3. Flat directory resolver

If you want to use a (flat) filesystem directory as a repository, simply type:

Example 28.24. Flat repository resolver

build.gradle

```
repositories {  
    flatDir name: 'localRepository', dirs: 'lib'  
    flatDir dirs: ['lib1', 'lib2']  
}
```

This adds repositories which look into one or more directories for finding dependencies. If you only work with flat directory resolvers you don't need to set all attributes of a dependency. See [Section 28.3.7, “Optional attributes”](#)

28.5.4. More about preconfigured repositories

The methods above for creating preconfigured repositories share some common behavior. For all of them, defining a name for the repository is optional. If no name is defined a default name is calculated, depending on the type of the repository. You might want to assign a name, if you want to access the declared repository. For example if you want to use it also for uploading your project artifacts. An explicit name might also be helpful when studying the debug output.

The values passed as arguments to the repository methods can be of any type, not just String. The value that is actually used, is the `toString` result of the argument object.

28.5.5. Cache

When Gradle downloads dependencies from remote repositories it stores them in a local cache located at `US`. When Gradle downloads dependencies from one of its predefined local resolvers (e.g. Flat Directory resolver), the cache is not used as an intermediate storage for dependency artifacts. The cache is always used for caching module descriptors.

28.5.6. More about Ivy resolvers

Gradle, thanks to Ivy under its hood, is extremely flexible regarding repositories:

- There are many options for the protocol to communicate with the repository (e.g. filesystem, http, ssh, ...)
- Each repository can have its own layout.

Let's say, you declare a dependency on the `junit:junit:3.8.2` library. Now how does Gradle find it in the repositories? Somehow the dependency information has to be mapped to a path. In contrast to Maven, where this path is fixed, with Gradle you can define a pattern that defines what the path will look like. Here are some examples: ^[22]

```
// Maven2 layout (if a repository is marked as Maven2 compatible, the organization (group) is not used)
someroot/[organisation]/[module]/[revision]/[module]-[revision].[ext]

// Typical layout for an ivy repository (the organization is not split into subfolder)
someroot/[organisation]/[module]/[revision]/[type]/[artifact].[ext]

// Simple layout (the organization is not used, no nested folders.)
someroot/[artifact]-[revision].[ext]
```

To add any kind of repository (you can pretty easy write your own ones) you can do:

Example 28.25. Definition of a custom repository

build.gradle

```
repositories {
    add(new org.apache.ivy.plugins.resolver.FileSystemResolver()) {
        name = 'repo'
        addIvyPattern "$projectDir/repo/[organisation]/[module]-ivy-[revision].xml"
        addArtifactPattern "$projectDir/repo/[organisation]/[module]-[revision](-[classifier])-[artifact].[ext]"
        descriptor = 'optional'
        checkmodified = true
    }
}
```

An overview of which Resolvers are offered by Ivy and thus also by Gradle can be found [here](#). With Gradle you just don't configure them via XML but directly via their API.

28.6. Strategies for transitive dependency management

Many projects rely on the [Maven2 repository](#). This is not without problems.

- The IBiblio repository can be down or has a very long response time.
- The `pom.xml`'s of many projects have wrong information (as one example, the `pom` of `commons-httpclient` declares JUnit as a runtime dependency).
- For many projects there is not one right set of dependencies (as more or less imposed by the `pom` format).

If your project relies on the IBiblio repository you are likely to need an additional custom repository, because:

- You might need dependencies that are not uploaded to IBiblio yet.
- You want to deal properly with wrong metadata in a IBiblio `pom.xml`.
- You don't want to expose people who want to build your project, to the downtimes or sometimes very long response times of IBiblio.

It is not a big deal to set-up a custom repository. ^[23] But it can be tedious, to keep it up to date. For a new version, you have always to create the new XML descriptor and the directories. And your custom repository is another infrastructure element which might have downtimes and needs to be updated. To enable historical builds, you need to keep all the past libraries and you need a backup. It is another layer of indirection.

Another source of information you have to lookup. All this is not really a big deal but in its sum it has an impact. Repository Manager like Artifactory or Nexus make this easier. But for example open source projects don't usually have a host for those products.

This is a reason why some projects prefer to store their libraries in their version control system. This approach is fully supported by Gradle. The libraries can be stored in a flat directory without any XML module descriptor files. Yet Gradle offers complete transitive dependency management. You can use either client module dependencies to express the dependency relations, or artifact dependencies in case a first level dependency has no transitive dependencies. People can check out such a project from svn and have everything necessary to build it.

If you are working with a distributed version control system like Git you probably don't want to use the version control system to store libraries as people check out the whole history. But even here the flexibility of Gradle can make your life easier. For example you can use a shared flat directory without XML descriptors and yet you can have full transitive dependency management as described above.

You could also have a mixed strategy. If your main concern is bad metadata in the `pom.xml` and maintaining custom XML descriptors, *Client Modules* offer an alternative. But you can of course still use Maven2 repo and your custom repository as a repository for *jars only* and still enjoy *transitive* dependency management. Or you can only provide client modules for pom's with bad metadata. For the jars and the correct pom's you still use the remote repository.

28.6.1. Implicit transitive dependencies

There is another way to deal with transitive dependencies *without* XML descriptor files. You can do this with Gradle, but we don't recommend it. We mention it for the sake of completeness and comparison with other build tools.

The trick is to use only artifact dependencies and group them in lists. That way you have somehow expressed, what are your first level dependencies and what are transitive dependencies (see [Section 28.3.7, "Optional attributes"](#)). But the draw-back is, that for the Gradle dependency management all dependencies are considered first level dependencies. The dependency reports don't show your real dependency graph and the `compile` task uses all dependencies, not just the first level dependencies. All in all, your build is less maintainable and reliable than it could be when using client modules. And you don't gain anything.

[19] JSR 294: Improved Modularity Support in the Java™ Programming Language, <http://jcp.org/en/jsr/detail?id=294>

[20] Gradle supports partial multiproject builds (see [Chapter 31, Multi-project Builds](#)).

[2 1]
<http://www.sonatype.com/books/maven-book/reference/pom-relationships-sect-project-relationships.html>

[22] At <http://ant.apache.org/ivy/history/latest-milestone/concept.html> you can learn more about ivy patterns.

[23] If you want to shield your project from the downtimes of IBiblio things get more complicated. You probably want to set-up a repository proxy for this. In an enterprise environment this is rather common. For an open source project it looks like overkill.

Artifact Management

29.1. Introduction

This chapter is about how you declare what are the artifacts of your project and how to work with them (e.g. upload them). We define the artifacts of the projects as the files the project want to provide to the outside world. This can be a library or a distribution or any other file. Usually artifacts are archives, but not necessarily. In the Maven world a project can provide only one artifact. With Gradle a project can provide as many artifacts as needed.

29.2. Artifacts and configurations

Like dependencies, artifacts are grouped by configurations. In fact, a configuration can contain both, artifacts and dependencies, at the same time. To assign an artifact to a configuration, you can write:

Example 29.1. Assignment of an artifact to a configuration

build.gradle

```
task myJar(type: Jar)

artifacts {
    archives myJar
}
```

What do you gain by assigning an artifact to a configuration? For each configuration (also for the custom ones added by you) Gradle provides the tasks `upload[ConfigurationName]` and `build[ConfigurationName]`.^[24] Execution of these tasks will build or upload the artifacts belonging to the respective configuration.

Table Table 18.5, “Java plugin - dependency configurations” shows the configurations added by the Java plugin. Two of the configurations are relevant for the usage with artifacts. The `archives` configuration is the standard configuration to assign your artifacts to. The Java plugin automatically assigns the default jar to this configuration. We will talk more about the `default` configuration in Section 29.4, “More about project libraries”. As with dependencies, you can declare as many custom configurations as you like and assign artifacts to them.

It is important to note that the custom archives you are creating as part of your build are not automatically assigned to any configuration. You have to explicitly do this assignment.

29.3. Uploading artifacts

We have said that there is a specific upload task for each configuration. But before you can do an upload, you have to configure the upload task and define where to upload. The repositories you have defined (as described in [Section 28.5, “Repositories”](#)) are not automatically used for uploading. In fact, some of those repositories allow only for artifacts downloading. Here is an example how you can configure the upload task of a configuration:

Example 29.2. Configuration of the upload task

build.gradle

```
repositories {
    flatDir(name: 'fileRepo', dirs: "$projectDir/repo")
}

uploadArchives {
    uploadDescriptor = false
    repositories {
        add project.repositories.fileRepo
        add(new org.apache.ivy.plugins.resolver.SshResolver()) {
            name = 'sshRepo'
            user = 'username'
            userPassword = 'pw'
            host = "http://repo.mycompany.com"
        }
    }
}
```

As you can see, you can either use a reference to an existing repository or create a new repository. As described in [Section 28.5.6, “More about Ivy resolvers”](#), you can use all the Ivy resolvers suitable for the purpose of uploading.

Uploading to a Maven repository is described in [Section 29.5, “Interacting with Maven repositories”](#).

29.4. More about project libraries

If your project is supposed to be used as a library, you need to define what are the artifacts of this library and what are the dependencies of these artifacts. The Java plugin adds a `default` configuration for this purpose. This configuration extends both the `archives` and the `runtime` configuration, with the implicit assumption that the `runtime` dependencies are the dependencies of the `archives` configuration. Of course this is fully customizable. You can add your own custom configuration or let the existing configurations extend from other configurations. You might have different group of artifacts which have a different set of dependencies. This mechanism is very powerful and flexible.

If someone wants to use your project as a library, she simply needs to declare on which configuration of the dependency to depend on. A Gradle dependency offers the `configuration` property to declare this. If this is not specified, the `default` configuration is used (see [Section 28.3.8, “Dependency configurations”](#)). Using your project as a library can either happen from within a multi-project build or by retrieving your project from a repository. In the latter case, an `ivy.xml` descriptor in the repository is supposed to contain all the

necessary information. If you work with Maven repositories you don't have the flexibility as described above. For how to publish to a Maven repository, see the section [Section 29.5, “Interacting with Maven repositories”](#).

29.5. Interacting with Maven repositories

29.5.1. Introduction

With Gradle you can deploy to remote Maven repositories or install to your local Maven repository. This includes all Maven metadata manipulation and works also for Maven snapshots. In fact, Gradle's deployment is 100 percent Maven compatible as we use the native Maven Ant tasks under the hood.

Deploying to a Maven repository is only half the fun if you don't have a pom. Fortunately Gradle can generate this pom for you using the dependency information it has.

29.5.2. Deploying to a Maven repository

Let's assume your project produces just the default jar file. Now you want to deploy this jar file to a remote Maven repository.

Example 29.3. Upload of file to remote Maven repository

build.gradle

```
usePlugin 'maven'

uploadArchives {
    repositories.mavenDeployer {
        repository(url: "file://localhost/tmp/myRepo/")
    }
}
```

That is all. Calling the `uploadArchives` task will generate the pom and deploys the artifact and the pom to the specified repository.

There is some more work to do if you need support for other protocols than `file`. In this case the native Maven code we delegate to needs additional libraries. Which libraries depend on the protocol you need. The available protocols and the corresponding libraries are listed in [Table 29.1, “Protocol jars for Maven deployment”](#) (those libraries have again transitive dependencies which have transitive dependencies). ^[25] For example to use the `ssh` protocol you can do:

Example 29.4. Upload of file via SSH

build.gradle

```
configurations {
    deployerJars
}

repositories {
    mavenCentral()
}

dependencies {
    deployerJars "org.apache.maven.wagon:wagon-ssh:1.0-beta-2"
}

uploadArchives {
    repositories.mavenDeployer {
        name = 'sshDeployer' // optional
        configuration = configurations.deployerJars
        repository(url: "scp://repos.mycompany.com/releases") {
            authentication(userName: "me", password: "myPassword")
        }
    }
}
```

There are many configuration options for the Maven deployer. The configuration is done via a Groovy builder. All the elements of this tree are Java beans. To configure the simple attributes you pass a map to the bean elements. To add another bean elements to its parent, you use a closure. In the example above *repository* and *authentication* are such bean elements. [Table 29.2, “Configuration elements of the MavenDeployer”](#) lists the available bean elements and a link to the javadoc of the corresponding class. In the javadoc you can see the possible attributes you can set for a particular element.

In Maven you can define repositories and optionally snapshot repositories. If no snapshot repository is defined, releases and snapshots are both deployed to the `repository` element. Otherwise snapshots are deployed to the `snapshotRepository` element.

Table 29.1. Protocol jars for Maven deployment

Protocol	Library
http	org.apache.maven.wagon:wagon-http:1.0-beta-2
ssh	org.apache.maven.wagon:wagon-ssh:1.0-beta-2
ssh-external	org.apache.maven.wagon:wagon-ssh-external:1.0-beta-2
scp	org.apache.maven.wagon:wagon-scp:1.0-beta-2
ftp	org.apache.maven.wagon:wagon-ftp:1.0-beta-2
webdav	org.apache.maven.wagon:wagon-webdav:1.0-beta-2
file	-

Table 29.2. Configuration elements of the MavenDeployer

Element	Javadoc
root	MavenDeployer
repository	org.apache.maven.artifact.ant.RemoteRepository
authentication	org.apache.maven.artifact.ant.Authentication
releases	org.apache.maven.artifact.ant.RepositoryPolicy
snapshots	org.apache.maven.artifact.ant.RepositoryPolicy
proxy	org.apache.maven.artifact.ant.Proxy
snapshotRepository	org.apache.maven.artifact.ant.RemoteRepository

29.5.3. Installing to the local repository

The Maven plugin adds an `install` task to your project. This task depends on all the archives task of the `artifacts` configuration. It installs those archives to your local Maven repository. If the default location for the local repository is redefined in a Maven `settings.xml`, this is considered by this task.

29.5.4. Maven Pom generation

The Maven Poms are automatically generated by Gradle. You can find the generated poms in the directory `<projectDir>/build/poms`. In many scenarios it just works and you don't have to do anything. But there are situations where you want to or have to customize the pom generation.

29.5.4.1. Changing non-dependency elements of the pom

You might want the artifact deployed to the maven repository to have a different version or name than the artifact generated by Gradle. To customize these you can do:

Example 29.5. Customization of pom

build.gradle

```
uploadArchives {
    repositories.mavenDeployer {
        repository(url: "file://localhost/tmp/myRepo/")
        pom.version = '1.0Maven'
        pom.artifactId = 'myMavenName'
    }
}
```

To learn about all the customizable attributes of a pom have a look here: [MavenPom](#). If you have more than one artifact to publish, things work differently. See [Section 29.5.4.2, “Multiple artifacts per project”](#).

To customize the settings for the Maven installer (see [Section 29.5.3, “Installing to the local repository”](#)), you can do:

Example 29.6. Customization of Maven installer

build.gradle

```
configure(install.repositories.mavenInstaller) {  
    pom.version = '1.0Maven'  
    pom.artifactId = 'myName'  
}
```

29.5.4.2. Multiple artifacts per project

Maven can only deal with one artifact per project. This is reflected in the structure of the Maven pom. We think there are many situations where it makes sense to have more than one artifact per project. In such a case you need to generate multiple poms. In such a case you have to explicitly declare each artifact you want to publish to a Maven repository. The [MavenDeployer](#) and the [MavenInstaller](#) both provide an API for this:

Example 29.7. Generation of multiple poms

build.gradle

```
uploadArchives {  
    repositories.mavenDeployer {  
        repository(url: "file://localhost/tmp/myRepo/")  
        addFilter('api') { artifact, file ->  
            artifact.name == 'api'  
        }  
        addFilter('service') { artifact, file ->  
            artifact.name == 'service'  
        }  
        pom('api').version = 'mySpecialMavenVersion'  
    }  
}
```

You need to declare a filter for each artifact you want to publish. This filter defines a boolean expression for which Gradle artifact it accepts. Each filter has a pom associated with it which you can configure. To learn more about this have a look at [GroovyPomFilterContainer](#) and its associated classes.

29.5.4.3. Dependency mapping

The Maven plugin configures the default mapping between the Gradle configurations added by the Java and War plugin and the Maven scopes. Most of the time you don't need to touch this and you can safely skip this section. The mapping works like the following. You can map a configuration to one and only one scope. Different configurations can be mapped to one or different scopes. One can assign also a priority to a particular configuration-to-scope mapping. Have a look at [Conf2ScopeMappingContainer](#) to learn more. To access the mapping configuration you can say:

Example 29.8. Accessing a mapping configuration

build.gradle

```
task mappings << {  
    println conf2ScopeMappings.mappings  
}
```

Gradle exclude rules are converted to Maven excludes if possible. Such a conversion is possible if in the Gradle exclude rule the group as well as the module name is specified (as Maven needs both in contrast to Ivy). Right now excludes-per-configuration are not converted to the Maven Pom.

29.5.4.4. Planned future features

We plan support for excludes-per-configuration. We also plan support for the new Ivy *override* element, which corresponds to the *dependencyManagement* element of a Maven pom. Last but not least we want to make the customization more powerful, by enabling to add custom dependency elements to the pom and remove/modify auto-generated ones.

[24] To be exact, the Base plugin provides those tasks. The BasePlugin is automatically applied, if you use the Java plugin.

[25] It is planned for a future release to provide out-of-the-box support for this

The Build Lifecycle

We said earlier, that the core of Gradle is a language for dependency based programming. In Gradle terms this means that you can define tasks and dependencies between tasks. Gradle guarantees that these tasks are executed in the order of their dependencies, and that each task is executed only once. Those tasks form a Directed Acyclic Graph. There are build tools that build up such a dependency graph as they execute their tasks. Gradle builds the complete dependency graph *before* any task is executed. This lies at the heart of Gradle and makes many things possible which would not be possible otherwise.

Your build scripts configure this dependency graph. Therefore they are strictly speaking *build configuration scripts*.

30.1. Build phases

A Gradle build has three distinct phases.

Initialization

Gradle supports single and multi-project builds. During the initialization phase, Gradle determines which projects are going to take part in the build, and creates a Project instance for each of these projects.

Configuration

The build scripts of *all* projects which are part of the build are executed. This configures the project objects.

Execution

Gradle determines the subset of the tasks, created and configured during the configuration phase, to be executed. The subset is determined by the task name arguments passed to the **gradle** command and the current directory. Gradle then executes each of the selected tasks.

30.2. Settings file

Beside the build script files, Gradle defines a settings file. The settings file is determined by Gradle via a naming convention. The default name for this file is `settings.gradle`. Later in this chapter we explain, how Gradle looks for a settings file.

The settings file gets executed during the initialization phase. A multiproject build must have a `settings.gr` file in the root project of the multiproject hierarchy. It is required because in the settings file it is defined, which projects are taking part in the multi-project build (see Chapter 31, Multi-project Builds). For a

single-project build, a settings file is optional. You might need it for example, to add libraries to your build script classpath (see [Chapter 34, Organizing Build Logic](#)). Let's first do some introspection with a single project build:

Example 30.1. Single project build

settings.gradle

```
println 'This is executed during the initialization phase.'
```

build.gradle

```
println 'This is executed during the configuration phase.'

task configured {
    println 'This is also executed during the configuration phase.'
}

task test << {
    println 'This is executed during the execution phase.'
}
```

Output of `gradle test`

```
> gradle test
This is executed during the initialization phase.
This is executed during the configuration phase.
This is also executed during the configuration phase.
:test
This is executed during the execution phase.

BUILD SUCCESSFUL

Total time: 1 secs
```

For a build script, the property access and method calls are delegated to a project object. Similarly property access and method calls within the settings file is delegated to a settings object. Have a look at [Settings](#).

30.3. Multi-project builds

A multi-project build is a build where you build more than one project during a single execution of Gradle. You have to declare the projects taking part in the multiproject build in the settings file. There is much more to say about multi-project builds in the chapter dedicated to this topic (see [Chapter 31, Multi-project Builds](#)).

30.3.1. Project locations

Multi-project builds are always represented by a tree with a single root. Each element in the tree represent a project. A project has a virtual and a physical path. The virtual path denotes the position of the project in the multi-project build tree. The project tree is created in the `settings.gradle` file. By default it is assumed that the location of the settings file is also the location of the root project. But you can redefine the location of the root project in the settings file.

30.3.2. Building the tree

In the settings file you can use a set of methods to build the project tree. Hierarchical and flat physical layouts get special support.

30.3.2.1. Hierarchical layouts

Example 30.2. Hierarchical layout

settings.gradle

```
include 'project1', 'project2', 'project2:child1'
```

The `include` method takes as an argument a relative virtual path to the root project. This relative virtual path is assumed to be equals to the relative physical path of the subproject to the root project. You only need to specify the leafs of the tree. Each parent path of the leaf project is assumed to be another subproject which obeys to the physical path assumption described above.

30.3.2.2. Flat layouts

Example 30.3. Flat layout

settings.gradle

```
includeFlat 'project3', 'project4'
```

The `includeFlat` method takes directory names as an argument. Those directories need to exist at the same level as the root project directory. The location of those directories are considered as child projects of the root project in the virtual multi-project tree.

30.3.3. Modifying elements of the project tree

The multi-project tree created in the settings file is made up of so called *project descriptors*. You can modify these descriptors in the settings file at any time. To access a descriptor you can do:

Example 30.4. Modification of elements of the project tree

settings.gradle

```
println rootProject.name  
println project(':projectA').name
```

Using this descriptor you can change the name, project directory and build file of a project.

Example 30.5. Modification of elements of the project tree

settings.gradle

```
rootProject.name = 'main'  
project(':projectA').projectDir = new File(settingsDir, '../my-project-a')  
project(':projectA').buildFileName = 'projectA.gradle'
```

Have a look at [ProjectDescriptor](#) for more details.

30.4. Initialization

How does Gradle know whether to do a single or multiproject build? If you trigger a multiproject build from the directory where the settings file is, things are easy. But Gradle also allows you to execute the build from within any subproject taking part in the build. ^[26] If you execute Gradle from within a project that has no `settings` file, Gradle does the following:

- It searches for a `settings.gradle` in a directory called `master` which has the same nesting level as the current dir.
- If no `settings.gradle` is found, it searches the parent directories for the existence of a `settings` file.
- If no `settings.gradle` file is found, the build is executed as a single project build.
- If a `settings.gradle` file is found, Gradle checks if the current project is part of the multiproject hierarchy defined in the found `settings.gradle` file. If not, the build is executed as a single project build. Otherwise a multiproject build is executed.

What is the purpose of this behavior? Somehow Gradle has to find out, whether the project you are into, is a subproject of a multiproject build or not. Of course, if it is a subproject, only the subproject and its dependent projects are build. But Gradle needs to create the build configuration for the whole multiproject build (see [Chapter 31, *Multi-project Builds*](#)). Via the `-u` command line option, you can tell Gradle not to look in the parent hierarchy for a `settings.gradle` file. The current project is then always build as a single project build. If the current project contains a `settings.gradle` file, the `-u` option has no meaning. Such a build is always executed as:

- a single project build, if the `settings.gradle` file does not define a multiproject hierarchy
- a multiproject build, if the `settings.gradle` file does define a multiproject hierarchy.

The auto search for a settings file does only work for multi-project builds with a physical hierarchical or flat layout. For a flat layout you must additionally obey to the naming convention described above. Gradle supports arbitrary physical layouts for a multiproject build. But for such arbitrary layouts you need to execute the build from the directory where the settings file is located. For how to run partial builds from the root see [Section 31.4, “Running tasks by their absolute path”](#). In our next release we want to enable partial builds from subprojects by specifying the location of the settings file as a command line parameter. Gradle creates Project objects for every project taking part in the build. For a single project build this is only one project. For a multi-project build these are the projects specified in Settings object (plus the root project). Each project object has by default a name equals to the name of its top level folder. Every project except the root project has a parent project and might have child projects.

30.5. Configuration and execution of a single project build

For a single project build, the workflow of the *after initialization* phases are pretty simple. The build script is executed against the project object that was created during the initialization phase. Then Gradle looks for tasks with names equal to those passed as command line arguments. If these task names exist, they are executed as a separate build in the order you have passed them. The configuration and execution for multi-project builds is discussed in [Chapter 31, *Multi-project Builds*](#).

30.6. Responding to the lifecycle in the build script

Your build script can receive notifications as the build progresses through its lifecycle. These notifications generally take 2 forms: You can either implement a particular listener interface, or you can provide a closure to execute when the notification is fired. The examples below use closures. For details on how to use the listener interfaces, refer to the API documentation.

30.6.1. Project evaluation

You can receive a notification immediately before and after a project is evaluated. This can be used to do things like performing additional configuration once all the definitions in a build script have been applied, or for some custom logging or profiling.

Below is an example which adds a `test` task to each project with the `hasTests` property set to true.

Example 30.6. Adding of test task to each project which has certain property set

build.gradle

```
allprojects {
    afterEvaluate { project ->
        if (project.hasTests) {
            println "Adding test task to $project"
            project.task('test') << {
                println "Running tests for $project"
            }
        }
    }
}
```

projectA.gradle

```
hasTests = true
```

Output of `gradle -q test`

```
> gradle -q test
Adding test task to project ':projectA'
Running tests for project ':projectA'
```

This example uses method `Project.afterEvaluate()` to add a closure which is executed after the project is evaluated.

It is also possible to receive notifications when any project is evaluated. This example performs some custom logging of project evaluation. Notice that the `afterProject` notification is received regardless of whether the project evaluates successfully or fails with an exception.

Example 30.7. Notifications

build.gradle

```
gradle.afterProject {project, exception ->
    if (exception) {
        println "Evaluation of $project FAILED"
    } else {
        println "Evaluation of $project succeeded"
    }
}
```

Output of `gradle -q test`

```
> gradle -q test
Evaluation of root project 'buildProjectEvaluateEvents' succeeded
Evaluation of project ':projectA' succeeded
Evaluation of project ':projectB' FAILED
```

You can also add a `ProjectEvaluationListener` to the `Gradle` to receive these events.

30.6.2. Task creation

You can receive a notification immediately after a task is added to a project. This can be used to set some default values or add behaviour before the task is made available in the build file.

The following example sets the `srcDir` property of each task as it is created.

Example 30.8. Setting of certain property to all tasks

build.gradle

```
tasks.whenTaskAdded { task ->
    task.srcDir = 'src/main/java'
}

task a

println "source dir is $a.srcDir"
```

Output of `gradle -q a`

```
> gradle -q a
source dir is src/main/java
```

You can also add a `TaskAction` to a `TaskContainer` to receive these events.

30.6.3. Task execution graph ready

You can receive a notification immediately after the task execution graph has been populated. We have seen this already in [Section 4.11, “Configure by DAG”](#).

You can also add a `TaskExecutionGraphListener` to the `TaskExecutionGraph` to receive these events.

30.6.4. Task execution

You can receive a notification immediately before and after any task is executed.

The following example logs the start and end of each task execution. Notice that the `afterTask` notification is received regardless of whether the task completes successfully or fails with an exception.

Example 30.9. Logging of start and end of each task execution

build.gradle

```
task ok

task broken(dependsOn: ok) << {
    throw new RuntimeException('broken')
}

gradle.taskGraph.beforeTask { task ->
    println "executing $task ..."
}

gradle.taskGraph.afterTask { task, exception ->
    if (exception) {
        println "FAILED"
    }
    else {
        println "done"
    }
}
```

Output of `gradle -q broken`

```
> gradle -q broken
executing task ':ok' ...
done
executing task ':broken' ...
FAILED
```

You can also use a [TaskExecutionListener](#) to the [TaskExecutionGraph](#) to receive these events.

[26] Gradle supports partial multiproject builds (see [Chapter 31, Multi-project Builds](#)).

Multi-project Builds

The powerful support for multi-project builds is one of Gradle's unique selling points. This topic is also the most intellectually challenging.

31.1. Cross project configuration

Let's start with a very simple multi-project build. After all Gradle is a general purpose build tool at its core, so the projects don't have to be java projects. Our first examples are about marine life.

31.1.1. Defining common behavior

We have the following project tree. This is a multi-project build with a root project `water` and a subproject `bluewhale`.

Example 31.1. Multi-project tree - water & bluewhale projects

Build layout

```
water/  
  build.gradle  
  settings.gradle  
  bluewhale/
```

Note: The code for this example can be found at `samples/userguide/multiproject/firstExample`

`settings.gradle`

```
include 'bluewhale'
```

And where is the build script for the `bluewhale` project? In Gradle build scripts are optional. Obviously for a single project build, a project without a build script doesn't make much sense. For multiproject builds the situation is different. Let's look at the build script for the `water` project and execute it:

Example 31.2. Build script of water (parent) project

build.gradle

```
Closure cl = { task -> println "I'm ${task.project.name}" }
task hello << cl
project(':bluewhale') {
    task hello << cl
}
```

Output of `gradle -q hello`

```
> gradle -q hello
I'm water
I'm bluewhale
```

Gradle allows you to access any project of the multi-project build from any build script. The Project API provides a method called `project()`, which takes a path as an argument and returns the Project object for this path. The capability to configure a project build from any build script we call *cross project configuration*. Gradle implements this via *configuration injection*.

We are not that happy with the build script of the `water` project. It is inconvenient to add the task explicitly for every project. We can do better. Let's first add another project called `krill` to our multi-project build.

Example 31.3. Multi-project tree - water, bluewhale & krill projects

Build layout

```
water/
  build.gradle
  settings.gradle
  bluewhale/
  krill/
```

Note: The code for this example can be found at `samples/userguide/multiproject/addKrill/w`

settings.gradle

```
include 'bluewhale', 'krill'
```

Now we rewrite the `water` build script and boil it down to a single line.

Example 31.4. Water project build script

build.gradle

```
allprojects {  
    task hello << { task -> println "I'm $task.project.name" }  
}
```

Output of **gradle -q hello**

```
> gradle -q hello  
I'm water  
I'm bluewhale  
I'm krill
```

Is this cool or is this cool? And how does this work? The Project API provides a property `allprojects` which returns a list with the current project and all its subprojects underneath it. If you call `allprojects` with a closure, the statements of the closure are delegated to the projects associated with `allprojects`. You could also do an iteration via `allprojects.each`, but that would be more verbose.

Other build systems use inheritance as the primary means for defining common behavior. We also offer inheritance for projects as you will see later. But Gradle uses configuration injection as the usual way of defining common behavior. We think it provides a very powerful and flexible way of configuring multiproject builds.

31.2. Subproject configuration

The Project API also provides a property for accessing the subprojects only.

31.2.1. Defining common behavior

Example 31.5. Defining common behaviour of all projects and subprojects

build.gradle

```
allprojects {  
    task hello << {task -> println "I'm $task.project.name" }  
}  
subprojects {  
    hello << {println "- I depend on water"}  
}
```

Output of **gradle -q hello**

```
> gradle -q hello  
I'm water  
I'm bluewhale  
- I depend on water  
I'm krill  
- I depend on water
```

31.2.2. Adding specific behavior

You can add specific behavior on top of the common behavior. Usually we put the project specific behavior in the build script of the project where we want to apply this specific behavior. But as we have already seen, we don't have to do it this way. We could add project specific behavior for the `bluewhale` project like this:

Example 31.6. Defining specific behaviour for particular project

build.gradle

```
allprojects {
    task hello << {task -> println "I'm $task.project.name" }
}
subprojects {
    hello << {println "- I depend on water"}
}
project(':bluewhale').hello << {
    println "I'm the largest animal that has ever lived on this planet."
}
```

Output of `gradle -q hello`

```
> gradle -q hello
I'm water
I'm bluewhale
- I depend on water
I'm the largest animal that has ever lived on this planet.
I'm krill
- I depend on water
```

As we have said, we usually prefer to put project specific behavior into the build script of this project. Let's refactor and also add some project specific behavior to the `krill` project.

Example 31.7. Defining specific behaviour for project krill

Build layout

```
water/  
  build.gradle  
  settings.gradle  
  bluewhale/  
    build.gradle  
  krill/  
    build.gradle
```

Note: The code for this example can be found at [samples/userguide/multiproject/spreadSpec](#)

settings.gradle

```
include 'bluewhale', 'krill'
```

bluewhale/build.gradle

```
hello.doLast { println "- I'm the largest animal that has ever lived on this planet." }
```

krill/build.gradle

```
hello.doLast {  
    println "- The weight of my species in summer is twice as heavy as all human beings"  
}
```

build.gradle

```
allprojects {  
    task hello << {task -> println "I'm $task.project.name" }  
}  
subprojects {  
    hello << {println "- I depend on water"}  
}
```

Output of `gradle -q hello`

```
> gradle -q hello  
I'm water  
I'm bluewhale  
- I depend on water  
- I'm the largest animal that has ever lived on this planet.  
I'm krill  
- I depend on water  
- The weight of my species in summer is twice as heavy as all human beings.
```

31.2.3. Project filtering

To show more of the power of configuration injection, let's add another project called `tropicalFish` and add more behavior to the build via the build script of the `water` project.

31.2.3.1. Filtering by name

Example 31.8. Adding custom behaviour to some projects (filtered by project name)

Build layout

```
water/  
  build.gradle  
  settings.gradle  
bluewhale/  
  build.gradle  
krill/  
  build.gradle  
tropicalFish/
```

Note: The code for this example can be found at `samples/userguide/multiproject/addTropica`

settings.gradle

```
include 'bluewhale', 'krill', 'tropicalFish'
```

build.gradle

```
allprojects {  
    task hello << {task -> println "I'm $task.project.name" }  
}  
subprojects {  
    hello << {println "- I depend on water"}  
}  
configure(subprojects.findAll {it.name != 'tropicalFish'}) {  
    hello << {println '- I love to spend time in the arctic waters.'}  
}
```

Output of `gradle -q hello`

```
> gradle -q hello  
I'm water  
I'm bluewhale  
- I depend on water  
- I love to spend time in the arctic waters.  
- I'm the largest animal that has ever lived on this planet.  
I'm krill  
- I depend on water  
- I love to spend time in the arctic waters.  
- The weight of my species in summer is twice as heavy as all human beings.  
I'm tropicalFish  
- I depend on water
```

The `configure()` method takes a list as an argument and applies the configuration to the projects in this list.

31.2.3.2. Filtering by properties

Using the project name for filtering is one option. Using dynamic project properties is another.

Example 31.9. Adding custom behaviour to some projects (filtered by project properties)

Build layout

```

water/
  build.gradle
  settings.gradle
bluewhale/
  build.gradle
krill/
  build.gradle
tropicalFish/
  build.gradle

```

Note: The code for this example can be found at [samples/userguide/multiproject/tropicalWi](#)

settings.gradle

```
include 'bluewhale', 'krill', 'tropicalFish'
```

bluewhale/build.gradle

```

arctic = true
hello.doLast { println "- I'm the largest animal that has ever lived on this planet." }

```

krill/build.gradle

```

arctic = true
hello.doLast {
    println "- The weight of my species in summer is twice as heavy as all human beings"
}

```

tropicalFish/build.gradle

```
arctic = false
```

build.gradle

```

allprojects {
    task hello << {task -> println "I'm $task.project.name" }
}
subprojects {
    hello {
        doLast {println "- I depend on water"}
        afterEvaluate { Project project ->
            if (project.arctic){ doLast {
                println '- I love to spend time in the arctic waters.' }
            }
        }
    }
}
}

```

Output of `gradle -q hello`

```
> gradle -q hello
I'm water
I'm bluewhale
- I depend on water
- I'm the largest animal that has ever lived on this planet.
- I love to spend time in the arctic waters.
I'm krill
- I depend on water
- The weight of my species in summer is twice as heavy as all human beings.
- I love to spend time in the arctic waters.
I'm tropicalFish
- I depend on water
```

In the build file of the `water` project we use an `afterEvaluate` notification. This means that the closure we are passing gets evaluated *after* the build scripts of the subproject are evaluated. As the property `arctic` is set in those build scripts, we have to do it this way. You will find more on this topic in [Section 31.6, “Dependencies - Which dependencies?”](#)

31.3. Execution rules for multi-project builds

When we have executed the `hello` task from the root project `dir` things behaved in an intuitive way. All the `h` tasks of the different projects were executed. Let's switch to the `bluewhale` `dir` and see what happens if we execute Gradle from there.

Example 31.10. Running build from subproject

Output of `gradle -q hello`

```
> gradle -q hello
I'm bluewhale
- I depend on water
- I'm the largest animal that has ever lived on this planet.
- I love to spend time in the arctic waters.
```

The basic rule behind Gradle's behavior is simple. Gradle looks down the hierarchy, starting with the *current dir*, for tasks with the name `hello` and executes them. One thing is very important to note. Gradle *always* evaluates *every* project of the multi-project build and creates all existing task objects. Then, according to the task name arguments and the current `dir`, Gradle filters the tasks which should be executed. Because of Gradle's cross project configuration *every* project has to be evaluated before *any* task gets executed. We will have a closer look at this in the next section. Let's now have our last marine example. Let's add a task to `bluewhale` and `krill`.

Example 31.11. Evaluation and execution of projects

bluewhale/build.gradle

```
arctic = true
hello << { println "- I'm the largest animal that has ever lived on this planet." }

task distanceToIceberg << {
    println '20 nautical miles'
}
```

krill/build.gradle

```
arctic = true
hello << { println "- The weight of my species in summer is twice as heavy as all human" }

task distanceToIceberg << {
    println '5 nautical miles'
}
```

Output of **gradle -q distanceToIceberg**

```
> gradle -q distanceToIceberg
20 nautical miles
5 nautical miles
```

Here the output without the `-q` option:

Example 31.12. Evaluation and execution of projects

Output of **gradle distanceToIceberg**

```
> gradle distanceToIceberg
:bluewhale:distanceToIceberg
20 nautical miles
:krill:distanceToIceberg
5 nautical miles

BUILD SUCCESSFUL

Total time: 1 secs
```

The build is executed from the `water` project. Neither `water` nor `tropicalFish` have a task with the name `distanceToIceberg`. Gradle does not care. The simple rule mentioned already above is: Execute all tasks down the hierarchy which have this name. Only complain if there is *no* such task!

31.4. Running tasks by their absolute path

As we have seen, you can run a multi-project build by entering any subproject dir and execute the build from there. All matching task names of the project hierarchy starting with the current dir are executed. But Gradle also offers to execute tasks by their absolute path (see also [Section 31.5, "Project and task paths"](#)):

Example 31.13. Running tasks by their absolute path

Output of `gradle -q :hello :krill:hello hello`

```
> gradle -q :hello :krill:hello hello
I'm water
I'm krill
- I depend on water
- The weight of my species in summer is twice as heavy as all human beings.
- I love to spend time in the arctic waters.
I'm tropicalFish
- I depend on water
```

The build is executed from the `tropicalFish` project. We execute the `hello` tasks of the `water`, the `krill` and the `tropicalFish` project. The first two tasks are specified by their absolute path, the last task is executed on the name matching mechanism described above.

31.5. Project and task paths

A project path has the following pattern: It starts always with a colon, which denotes the root project. The root project is the only project in a path that is not specified by its name. The path `:bluewhale` corresponds to the file system path `water/project` in the case of the example above.

The path of a task is simply its project path plus the task name. For example `:bluewhale:hello`. Within a project you can address a task of the same project just by its name. This is interpreted as a relative path.

Originally Gradle has used the `'/'` character as a natural path separator. With the introduction of directory tasks (see [Section 11.1, “Directory creation”](#)) this was no longer possible, as the name of the directory task contains the `'/'` character.

31.6. Dependencies - Which dependencies?

The examples from the last section were special, as the projects had no *Execution Dependencies*. They had only *Configuration Dependencies*. Here is an example where this is different:

31.6.1. Execution dependencies

31.6.1.1. Dependencies and execution order

Example 31.14. Dependencies and execution order

Build layout

```
messages/  
  settings.gradle  
  consumer/  
    build.gradle  
  producer/  
    build.gradle
```

Note: The code for this example can be found at `samples/userguide/multiproject/dependencies`

settings.gradle

```
include 'consumer', 'producer'
```

consumer/build.gradle

```
task action << {  
    println "Consuming message: " + System.getProperty('org.gradle.message')  
}
```

producer/build.gradle

```
task action << {  
    println "Producing message:"  
    System.setProperty('org.gradle.message', 'Watch the order of execution.')  
}
```

Output of `gradle -q action`

```
> gradle -q action  
Consuming message: null  
Producing message:
```

This did not work out. If nothing else is defined, Gradle executes the task in alphanumeric order. Therefore `:consumer:action` is executed before `:producer:action`. Let's try to solve this with a hack and rename the producer project to `aProducer`.

Example 31.15. Dependencies and execution order

Build layout

```
messages/  
  settings.gradle  
  aProducer/  
    build.gradle  
  consumer/  
    build.gradle
```

settings.gradle

```
include 'consumer', 'aProducer'
```

aProducer/build.gradle

```
task action << {  
    println "Producing message:"  
    System.setProperty('org.gradle.message', 'Watch the order of execution.')  
}
```

consumer/build.gradle

```
task action << {  
    println "Consuming message: " + System.getProperty('org.gradle.message')  
}
```

Output of **gradle -q action**

```
> gradle -q action  
Producing message:  
Consuming message: Watch the order of execution.
```

Now we take the air out of this hack. We simply switch to the `consumer` dir and execute the build.

Example 31.16. Dependencies and execution order

Output of **gradle -q action**

```
> gradle -q action  
Consuming message: null
```

For Gradle the two `action` tasks are just not related. If you execute the build from the `messages` project Gradle executes them both because they have the same name and they are down the hierarchy. In the last example only one `action` was down the hierarchy and therefore it was the only task that got executed. We need something better than this hack.

31.6.1.2. Declaring dependencies

Example 31.17. Declaring dependencies

Build layout

```
messages/  
  settings.gradle  
  consumer/  
    build.gradle  
  producer/  
    build.gradle
```

Note: The code for this example can be found at `samples/userguide/multiproject/dependencies`

settings.gradle

```
include 'consumer', 'producer'
```

consumer/build.gradle

```
dependsOn(':producer')  
  
task action << {  
    println "Consuming message: " + System.getProperty('org.gradle.message')  
}
```

producer/build.gradle

```
task action << {  
    println "Producing message:"  
    System.setProperty('org.gradle.message', 'Watch the order of execution.')  
}
```

Output of `gradle -q action`

```
> gradle -q action  
Producing message:  
Consuming message: Watch the order of execution.
```

Running this from the `consumer` directory gives:

Example 31.18. Declaring dependencies

Output of `gradle -q action`

```
> gradle -q action  
Producing message:  
Consuming message: Watch the order of execution.
```

We have now declared that the `consumer` project has an *execution dependency* on the `producer` project. For Gradle declaring *execution dependencies* between *projects* is syntactic sugar. Under the hood Gradle creates task dependencies out of them. You can also create cross project tasks dependencies manually by using the absolute path of the tasks.

31.6.1.3. The nature of project dependencies

Let's change the naming of our tasks and execute the build.

Example 31.19. Project execution dependencies

consumer/build.gradle

```
dependsOn(':producer')

task consume << {
    println "Consuming message: " + System.getProperty('org.gradle.message')
}
```

producer/build.gradle

```
task produce << {
    println "Producing message:"
    System.setProperty('org.gradle.message', 'Watch the order of execution.')
}
```

Output of **gradle -q consume**

```
> gradle -q consume
Consuming message: null
```

Oops. Why does this not work? The `dependsOn` command is created for projects with a common lifecycle. Provided you have two Java projects where one depends on the other. If you trigger a compile for the dependent project you don't want that *all* tasks of the other project get executed. Therefore a `dependsOn` creates dependencies between tasks with equal names. To deal with the scenario above you would do the following:

Example 31.20. Cross project task dependencies

consumer/build.gradle

```
task consume(dependsOn: ':producer:produce') << {
    println "Consuming message: " + System.getProperty('org.gradle.message')
}
```

producer/build.gradle

```
task produce << {
    println "Producing message:"
    System.setProperty('org.gradle.message', 'Watch the order of execution.')
}
```

Output of **gradle -q consume**

```
> gradle -q consume
Producing message:
Consuming message: Watch the order of execution.
```

31.6.2. Configuration time dependencies

Let's have one more example with our producer-consumer build before we enter *Java* land. We add a property to the producer project and create now a configuration time dependency from consumer on producer.

Example 31.21. Configuration time dependencies

consumer/build.gradle

```
key = 'unknown'
if (project(':producer').hasProperty('key')) {
    key = project(':producer').key
}
task consume(dependsOn: ':producer:produce') << {
    println "Consuming message from key '$key': " + System.getProperty(key)
}
```

producer/build.gradle

```
key = 'org.gradle.message'

task produce << {
    println "Producing message:"
    System.setProperty(key, 'Watch the order of execution.')
}
```

Output of `gradle -q consume`

```
> gradle -q consume
Producing message:
Consuming message from key 'unknown': null
```

The default *evaluation* order of the projects is alphanumeric (for the same nesting level). Therefore the `consumer` project is evaluated before the `producer` project and the `key` value of the producer is set *after* it is read by the consumer project. Gradle offers a solution for this.

Example 31.22. Configuration time dependencies - `evaluationDependsOn`

consumer/build.gradle

```
evaluationDependsOn(':producer')

key = 'unknown'
if (project(':producer').hasProperty('key')) {
    key = project(':producer').key
}
task consume(dependsOn: ':producer:produce') << {
    println "Consuming message from key '$key': " + System.getProperty(key)
}
```

Output of `gradle -q consume`

```
> gradle -q consume
Producing message:
Consuming message from key 'org.gradle.message': Watch the order of execution.
```

The command `evaluationDependsOn` triggers the evaluation of `producer` *before* `consumer` is evaluated. The example is a bit contrived for the sake of showing the mechanism. In *this* case there would be an easier solution by reading the key property at execution time.

Example 31.23. Configuration time dependencies

consumer/build.gradle

```
task consume(dependsOn: ':producer:produce') << {  
    String key = project(':producer').key  
    println "Consuming message from key '$key': " + System.getProperty(key)  
}
```

Output of `gradle -q consume`

```
> gradle -q consume  
Producing message:  
Consuming message from key 'org.gradle.message': Watch the order of execution.
```

Configuration dependencies are very different to execution dependencies. Configuration dependencies are between projects whereas execution dependencies are always resolved to task dependencies. Another difference is that always all projects are configured, even when you start the build from a subproject. The default configuration order is top down, which is usually what is needed.

On the same nesting level the configuration order depends on the alphanumeric position. The most common use case is to have multi-project builds that share a common lifecycle (e.g. all projects use the Java plugin). If you declare with `dependsOn` a *execution dependency* between different projects, the default behavior of this method is to create also a *configuration* dependency between the two projects. Therefore it is likely that you don't have to define configuration dependencies explicitly.

31.6.3. Real life examples

Gradle's multi-project features are driven by real life use cases. The first example for describing such a use case, consists of two webapplication projects and a parent project that creates a distribution out of them. ^[27] For the example we use only one build script and do *cross project configuration*.

Example 31.24. Dependencies - real life example - crossproject configuration

Build layout

```
webDist/  
  settings.gradle  
  build.gradle  
  date/  
    src/main/java/org/gradle/sample/DateServlet.java  
  hello/  
    src/main/java/org/gradle/sample/HelloServlet.java
```

Note: The code for this example can be found at [samples/userguide/multiproject/dependencies](#)

settings.gradle

```
include 'date', 'hello'
```

build.gradle

```
dependsOnChildren()  
  
allprojects {  
    usePlugin('java')  
    group = 'org.gradle.sample'  
    version = '1.0'  
}  
  
subprojects {  
    usePlugin('war')  
    repositories {  
        mavenCentral()  
    }  
    dependencies {  
        compile "javax.servlet:servlet-api:2.5"  
    }  
}  
  
task explodedDist(dependsOn: assemble) << {  
    File explodedDist = mkdir(buildDir, 'explodedDist')  
    subprojects.each {project ->  
        project.tasks.withType(Jar).each {archiveTask ->  
            copy {  
                from archiveTask.archivePath  
                into explodedDist  
            }  
        }  
    }  
}
```

We have an interesting set of dependencies. Obviously the `date` and `hello` task have a *configuration* dependency on `webDist`, as all the build logic for the webapp projects is injected by `webDist`. The *execution* dependency is in the other direction, as `webDist` depends on the build artifacts of `date` and `hello`. There is even a third dependency. `webDist` has a *configuration* dependency on `date` and `hello` because it needs to know the `archivePath`. But it asks for this information at *execution time*. Therefore we have no circular dependency.

Such and other dependency patterns are daily bread in the problem space of multi-project builds. If a build

system does not support such patterns, you either can't solve your problem or you need to do ugly hacks which are hard to maintain and massively afflict your productivity as a build master.

There is one more thing to note from the current example. We have used the command `dependOnChildren`. It is a convenience method and calls the `dependsOn` method of the parent project for every child project (not every sub project). It declares a execution dependency of `webDist` on `date` and `hello`.

Another use case would be a situation where the subprojects have a configuration *and* execution dependency on the parent project. This is the case when the parent project does configuration injection into its subprojects, and additionally produces something at execution time that is needed by its child projects (e.g. code generation). In this case the parent project would call the `childrenDependOnMe` method to create an execution dependency for the child projects. We might add an example for this in a future version of the user guide.

31.7. Project lib dependencies

What if one projects needs the jar produced by another project in its compile path. And not just the jar but also the transitive dependencies of this jar. Obviously this is a very common use case for Java multi-project builds. As already mentioned in [Section 28.3.4, "Project dependencies"](#), Gradle offers project lib dependencies for this.

Example 31.25. Project lib dependencies

Build layout

```
java/
  settings.gradle
  build.gradle
  api/
    src/main/java/org/gradle/sample/api/Person.java
    src/main/java/org/gradle/sample/apiImpl/PersonImpl.java
  services/
    personService/
      src/main/java/org/gradle/sample/services/PersonService.java
      src/test/java/org/gradle/sample/services/PersonServiceTest.java
  shared/
    src/main/java/org/gradle/sample/shared/Helper.java
```

Note: The code for this example can be found at [samples/userguide/multiproject/dependencies](#)

We have the projects `shared`, `api` and `personService`. `personService` has a lib dependency on the other two projects. `api` has a lib dependency on `shared`. ^[28]

Example 31.26. Project lib dependencies

settings.gradle

```
include 'api', 'shared', 'services:personService'
```

build.gradle

```
subprojects {
    repositories {
        mavenCentral()
    }
    usePlugin('java')
    group = 'org.gradle.sample'
    version = '1.0'
}

project(':api') {
    dependencies {
        compile project(':shared')
    }
}

project(':services:personService') {
    dependencies {
        compile project(':shared'), project(':api')
        testCompile "junit:junit:3.8.2"
    }
}

dependsOnChildren()
```

All the build logic is in the `build.gradle` of the root project. ^[29] A *lib* dependency is a special form of an execution dependency. It causes the other project to be build first and adds the jar with the classes of the other project to the classpath. It also add the dependencies of the other project to the classpath. So you can enter the `api` folder and trigger a **gradle compile**. First `shared` is build and then `api` is build. Project dependencies enable partial multi-project builds.

If you come from Maven land you might be perfectly happy with this. If you come from Ivy land, you might expect some more fine grained control. Gradle offers this to you:

Example 31.27. Fine grained control over dependencies

build.gradle

```
subprojects {
    usePlugin('java')
    group = 'org.gradle.sample'
    version = '1.0'
}

project(':api') {
    configurations {
        spi
    }
    dependencies {
        compile project(':shared')
    }
    task spiJar(type: Jar) {
        baseName = 'api-spi'
        confs = ['spi']
        fileSet() {
            include('org/gradle/sample/api/**')
        }
    }
}

project(':services:personService') {
    dependencies {
        compile project(':shared')
        compile project(path: ':api', configuration: 'spi')
        testCompile "junit:junit:3.8.2", project(':api')
    }
}
```

The Java plugin adds per default a jar to your project libraries which contains all the classes. In this example we create an *additional* library containing only the interfaces of the `api` project. We assign this library to a new *dependency configuration*. For the person service we declare that the project should be compiled only against the `api` interfaces but tested with all classes from `api`.

31.7.1. Disable the build of dependency projects.

Sometimes you don't want depended on projects to be built when doing a partial build. To disable the build of the depended on projects you can run gradle with the `-a` option.

31.8. Multi-Project Building and Testing

The `build` task of the Java plugin is typically used to compile, test, and perform code style checks (if the CodeQuality plugin is used) of a single project. In multi-project builds you may often want to do all of these tasks across a range of projects. The `buildNeeded` and `buildDependents` tasks can help with this.

Let's use the project structure shown in [Example 31.26, "Project lib dependencies"](#). In this example `:services:personService` depends on both `:api` and `:shared`. The `:api` project also depends on `:shared`.

Assume you are working on a single project, the `:api` project. You have been making changes, but have not built the entire project since performing a clean. You want to build any necessary supporting jars, but only perform code quality and unit tests on the project you have changed. The `build` task does this.

Example 31.28. Build and Test Single Project

Output of `gradle :api:build`

```
> gradle :api:build
:shared:compileJava
:shared:processResources
:shared:classes
:shared:jar
:shared:uploadDefaultInternal
:api:compileJava
:api:processResources
:api:classes
:api:jar
:api:assemble
:api:compileTestJava
:api:processTestResources
:api:testClasses
:api:test
:api:check
:api:build

BUILD SUCCESSFUL

Total time: 1 secs
```

While you are working in a typical development cycle repeatedly building and testing changes to a the `:api` project (knowing that you are only changing files in this one project), you may not want to even suffer the expense of `:shared:compile` checking to see what has changed in the `:shared` project. Adding the `-a` option will cause gradle to use cached jars to resolve any project lib dependencies and not try to re-build the depended on projects.

Example 31.29. Partial Build and Test Single Project

Output of `gradle -a :api:build`

```
> gradle -a :api:build
:api:compileJava
:api:processResources
:api:classes
:api:jar
:api:assemble
:api:compileTestJava
:api:processTestResources
:api:testClasses
:api:test
:api:check
:api:build

BUILD SUCCESSFUL

Total time: 1 secs
```

If you have just gotten the latest version of source from your version control system which included changes in other projects that `:api` depends on, you might want to not only build all the projects you depend on, but test them as well. The `buildNeeded` task also tests all the projects from the project lib dependencies of the `testRuntime` configuration.

Example 31.30. Build and Test Depended On Projects

Output of `gradle :api:buildNeeded`

```
> gradle :api:buildNeeded
:shared:compileJava
:shared:processResources
:shared:classes
:shared:jar
:shared:uploadDefaultInternal
:api:compileJava
:api:processResources
:api:classes
:api:jar
:api:assemble
:api:compileTestJava
:api:processTestResources
:api:testClasses
:api:test
:api:check
:api:build
:shared:assemble
:shared:compileTestJava
:shared:processTestResources
:shared:testClasses
:shared:test
:shared:check
:shared:build
:api:buildNeeded

BUILD SUCCESSFUL

Total time: 3.656 secs
```

You also might want to refactor some part of the `:api` project that is used in other projects. If you make these types of changes, it is not sufficient to test just the `:api` project, you also need to test all projects that depend on the `:api` project. The `buildDependents` task also tests all the projects that have a project lib dependency (in the `testRuntime` configuration) on the specified project.

Example 31.31. Build and Test Dependent Projects

Output of `gradle :api:buildDependents`

```
> gradle :api:buildDependents
:shared:compileJava
:shared:processResources
:shared:classes
:shared:jar
:shared:uploadDefaultInternal
:api:compileJava
:api:processResources
:api:classes
:api:jar
:api:assemble
:api:compileTestJava
:api:processTestResources
:api:testClasses
:api:test
:api:check
:api:build
:api:uploadDefaultInternal
:services:personService:compileJava
:services:personService:processResources
:services:personService:classes
:services:personService:jar
:services:personService:assemble
:services:personService:compileTestJava
:services:personService:processTestResources
:services:personService:testClasses
:services:personService:test
:services:personService:check
:services:personService:build
:api:buildDependents

BUILD SUCCESSFUL

Total time: 1 secs
```

Finally, you may want to build and test everything in all projects. If the root project has declared `dependsOn` (as this one does), then any task you run against the root project will cause that same named task to be run on all the children. So you can just run `gradle build` to build and test all projects.

31.9. Property and method inheritance

Properties and methods declared in a project are inherited to all its subprojects. This is an alternative to configuration injection. But we think that the model of inheritance does not reflect the problem space of multi-project builds very well. In a future edition of this user guide we might write more about this.

Method inheritance might be interesting to use as Gradle's *Configuration Injection* does not support methods yet (but will in a future release.).

You might be wondering why we have implemented a feature we obviously don't like that much. One reason is that it is offered by other tools and we want to have the check mark in a feature comparison :). And we like to offer our users a choice.

31.10. Summary

Writing this chapter was pretty exhausting and reading it might have a similar effect. Our final message for this chapter is that multi-project builds with Gradle are usually *not* difficult. There are six elements you need to remember: `allproject`, `subprojects`, `dependsOn`, `childrenDependOnMe`, `dependOnChildren` and project lib dependencies. [30] With those elements, and keeping in mind that Gradle has a distinct configuration and execution phase, you have already a lot of flexibility. But when you enter steep territory Gradle does not become an obstacle and usually accompanies and carries you to the top of the mountain.

[27] The real use case we had, was using <http://lucene.apache.org/solr>, where you need a separate war for each index your are accessing. That was one reason why we have created a distribution of webapps. The Resin servlet container allows us, to let such a distribution point to a base installation of the servlet container.

[28] `services` is also a project, but we use it just as a container. It has no build script and gets nothing injected by another build script.

[29] We do this here, as it makes the layout a bit easier. We usually put the project specific stuff into the buildscript of the respective projects.

[30] So we are well in the range of the 7 plus 2 Rule :)

Writing Custom Tasks

Gradle supports 2 types of task. One such type is the simple task, where you define the task with an action closure. We have seen these in [Chapter 4, *Build Script Basics*](#). For this type of task, the action closure determines the behaviour of the task. This type of task is good for implementing one-off tasks in your build script.

The other type of task is the enhanced task, where the behaviour is built into the task, and the task provides some properties which you can use to configure the behaviour. We have seen these in [Chapter 13, *More about Tasks*](#). Most Gradle plugins use enhanced tasks. With enhanced tasks, you don't need to implement the task behaviour as you do with simple tasks. You simply declare and configure the task using its properties. In this way, enhanced tasks let you reuse a piece of behaviour in many different places, potentially across different builds.

Gradle allows you to implement your own custom enhanced tasks. Implementing such a custom task in Gradle is easy. You can implement a custom task in pretty much any language you like, provided it ends up compiled to classes files. In our examples, we are going to use Groovy as the implementation language, but you could use, for example, Java or Scala.

There are several places where you can put the source for the task. Two convenient options are to add the task implementation to your build script, or to put the source in the `rootProjectDir/buildSrc/src/main` directory. Gradle will take care of compiling the task and making it available on the classpath of the build script. See [Chapter 34, *Organizing Build Logic*](#) for more details, and some other options. In our examples, we will put the task implementation in the build script, to keep things simple.

To implement a custom task, you extend `DefaultTask`.

Example 32.1. Defining a custom task

build.gradle

```
class HelloTask extends DefaultTask {  
}
```

This task doesn't do anything useful, so let's add some behaviour. To do so, we add a method to the task and mark it with the `TaskAction` annotation. Gradle will call the method when the task executes. You don't have to use a method to define the behaviour for the task. You could, for instance, call `doFirst()` or `doLast` with a closure in the task constructor to add behaviour.

Example 32.2. A hello world task

build.gradle

```
task hello(type: HelloTask)

class HelloTask extends DefaultTask {
    @TaskAction
    def printGreeting() {
        println 'hello from HelloTask'
    }
}
```

Output of **gradle -q hello**

```
> gradle -q hello
hello from HelloTask
```

Let's add a property to the task, so we can customize it.

Example 32.3. A customizable hello world task

build.gradle

```
task hello(type: HelloTask)

task greeting(type: HelloTask) {
    greeting = 'greetings from HelloTask'
}

class HelloTask extends DefaultTask {
    def String greeting = 'hello from HelloTask'

    @TaskAction
    def printGreeting() {
        println greeting
    }
}
```

Output of **gradle -q hello greeting**

```
> gradle -q hello greeting
hello from HelloTask
greetings from HelloTask
```

Writing Custom Plugins

A Gradle plugin packages up some reusable piece of build logic, which can be used across many different projects and builds. Gradle allows you to implement your own custom plugins, so you can reuse your build logic, and potentially share it with others.

You can implement a custom plugin in any language you like, provided the implementation ends up compiled as class files. For the examples here, we are going to use Groovy as the implementation language. You could use Java or Scala instead, if you wanted.

There are several places where you can put the source for the plugin. Two convenient options are to add the task implementation to your build script, or to put the source in the `rootProjectDir/buildSrc/src/main` directory. Gradle will take care of compiling the task and making it available on the classpath of the build script. See [Chapter 34, *Organizing Build Logic*](#) for more details, and some other options. In our examples, we will put the task implementation in the build script, to keep things simple.

To create a custom plugin, you need to write an implementation of `Plugin`. Gradle instantiates the plugin and calls the plugin instance's `use()` method when the plugin is used with a project. The project object is passed as a parameter, which the plugin can use to configure the project however it needs to. The following sample contains a greeting plugin, which adds a `hello` task to the project.

Example 33.1. A custom plugin

build.gradle

```
usePlugin(GreetingPlugin)

class GreetingPlugin implements Plugin {
    def void use(Project project, ProjectPluginsContainer projectPluginsHandler) {
        project.task('hello') << {
            println "Hello from the GreetingPlugin"
        }
    }
}
```

Output of `gradle -q hello`

```
> gradle -q hello
Hello from the GreetingPlugin
```

One thing to note is that only one instance of a given plugin is created for a given build. The same plugin

instance is used for all projects in the build.

Let's add a convention object to the project. Here we add a `greeting` property to the project, which allows you to configure the greeting.

Example 33.2. A custom plugin convention

`build.gradle`

```
usePlugin(GreetingPlugin)

greeting = 'Hi from Gradle'

class GreetingPlugin implements Plugin {
    def void use(Project project, ProjectPluginsContainer projectPluginsHandler) {
        project.convention.plugins.greeting = new GreetingPluginConvention()
        project.task('hello') << {
            println project.greeting
        }
    }
}

class GreetingPluginConvention {
    def String greeting = 'Hello from GreetingPlugin'
}
```

Output of `gradle -q hello`

```
> gradle -q hello
Hi from Gradle
```


Organizing Build Logic

Gradle offers a variety of ways to organize your build logic. First of all you can put your build logic directly in the action closure of a task. If a couple of tasks share the same logic you can extract this logic into a method. If multiple projects of a multi-project build share some logic you can define this method in the parent project. If the build logic gets too complex for being properly modeled by methods you want have an OO Model. ^[31] Gradle makes this very easy. Just drop your classes in a certain folder and Gradle automatically compiles them and puts them in the classpath of your build script.

34.1. Build sources

If you run Gradle, it checks for the existence of a folder called `buildSrc`. Just put your build source code in this folder and stick to the layout convention for a Java/Groovy project (see Table 18.4, “Java plugin - default project layout”). Gradle then automatically compiles and tests this code and puts it in the classpath of your build script. You don't need to provide any further instruction. For multi-project builds there can be only one `buildSrc` directory which has to be in the root project.

This is probably good enough for most of the cases. If you need more flexibility, you can provide a `build.gradle` and a `settings.gradle` file in the `buildSrc` folder. If you like, you can even have a multi-project build in there.

34.2. External dependencies for the build script

If your build script needs to use external libraries, you can add them to the script's classpath in the build script itself. You do this using the `buildscript()` method, passing in a closure which declares the build script classpath.

Example 34.1. Declaring external dependencies for the build script

`build.gradle`

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath group: 'commons-codec', name: 'commons-codec', version: '1.2'
    }
}
```

The closure passed to the `buildscript()` method configures a `ScriptHandler` instance. You declare the build script classpath by adding dependencies to the `classpath` configuration. This is the same way you declare, for example, the Java compilation classpath. You can use any of the dependency types described in [Section 28.3, “How to declare your dependencies”](#), except project dependencies.

Having declared the build script classpath, you can use the classes in your build script as you would any other classes on the classpath. The following example adds to the previous example, and uses classes from the build script classpath.

Example 34.2. A build script with external dependencies

build.gradle

```
import org.apache.commons.codec.binary.Base64

buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath group: 'commons-codec', name: 'commons-codec', version: '1.2'
    }
}

task encode << {
    def byte[] encodedString = new Base64().encode('hello world\n' as byte[])
    println new String(encodedString)
}
```

Output of `gradle -q encode`

```
> gradle -q encode
aGVsbG8gd29ybGQK
```

For multi-project builds, the dependencies declared in the a project's build script, are available to the build scripts of all sub-projects.

34.3. Ant optional dependencies

For reasons we don't fully understand yet, external dependencies are not picked up by Ant's optional tasks. But you can easily do it in another way. [\[32\]](#)

Example 34.3. Ant optional dependencies

build.gradle

```
configurations {
    ftpAntTask
}

dependencies {
    ftpAntTask("org.apache.ant:ant-commons-net:1.7.0") {
        module("commons-net:commons-net:1.4.1") {
            dependencies "oro:oro:2.0.8:jar"
        }
    }
}

task ftp << {
    ant {
        taskdef(name: 'ftp',
                classname: 'org.apache.tools.ant.taskdefs.optional.net.FTP',
                classpath: configurations.ftpAntTask.asPath)
        ftp(server: "ftp.apache.org", userid: "anonymous", password: "me@myorg.com") {
            fileset(dir: "htdocs/manual")
        }
    }
}
```

This is also nice example for the usage of client modules. The pom.xml in maven central for the ant-commons-net task does not provide the right information for this use case.

34.4. Summary

Gradle offers you a variety of ways of organizing your build logic. You can choose what is right for your domain and find the right balance between unnecessary indirections, and avoiding redundancy and a hard to maintain code base. It is our experience that even very complex custom build logic is rarely shared between different builds. Other build tools enforce a separation of this build logic into a separate project. Gradle spares you this unnecessary overhead and indirection.

[31] Which might range from a single class to something very complex.

[32] In fact, we think this is anyway the nicer solution. Only if your buildscript and Ant's optional task need the *same* library you would have to define it two times. In such a case it would be nice, if Ant's optional task would automatically pickup the classpath defined in the `gradlesettings`.

Initialization Scripts

Gradle provides a powerful mechanism to allow customizing the build based on the current environment. This mechanism also supports tools that wish to integrate with Gradle.

35.1. Basic usage

Initialization scripts (a.k.a. *init scripts*) are similar to other scripts in Gradle. These scripts, however, are run before the build starts. Here are several possible uses:

- Set up properties based on the current environment (such as a developer's machine vs. a continuous integration server).
- Supply personal information about the user to the build, such as repository or database authentication credentials.
- Define machine specific details, such as where JDKs are installed.
- Register build listeners. External tools that wish to listen to Gradle events might find this useful.

One main limitation of init scripts is that they cannot access classes in the `buildSrc` special module (see [Section 34.1, “Build sources”](#) for details of this feature).

There are two ways to use init scripts. Either put a file called `init.gradle` in `USER_HOME/.gradle`, or specify the file on the command line. The command line option is `-I` or `--init-script` followed by the path to the script. The command line option can appear more than once, each time adding another init script. If more than one init script is found they will all be executed. This allows for a tool to specify an init script and the user to put home in their home directory for defining the environment and both scripts will run when gradle is executed.

35.2. External dependencies for the init script

In [Section 34.2, “External dependencies for the build script”](#) it was explained how to add external dependencies to a build script. Init scripts can similarly have external dependencies defined. You do this using the `initscript()` method, passing in a closure which declares the init script classpath.

Example 35.1. Declaring external dependencies for an init script

init.gradle

```
initscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath group: 'org.apache.commons', name: 'commons-math', version: '2.0'
    }
}
```

The closure passed to the `initscript()` method configures a `ScriptHandler` instance. You declare the init script classpath by adding dependencies to the `classpath` configuration. This is the same way you declare, for example, the Java compilation classpath. You can use any of the dependency types described in [Section 28.3, “How to declare your dependencies”](#), except project dependencies.

Having declared the init script classpath, you can use the classes in your init script as you would any other classes on the classpath. The following example adds to the previous example, and uses classes from the init script classpath.

Example 35.2. An init script with external dependencies

init.gradle

```
import org.apache.commons.math.fraction.Fraction

initscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath group: 'org.apache.commons', name: 'commons-math', version: '2.0'
    }
}

println Fraction.ONE_FIFTH.multiply(2)
```

Output of `gradle --init-script init.gradle -q doNothing`

```
> gradle --init-script init.gradle -q doNothing
2 / 5
```

The Gradle Wrapper

Gradle is a new tool. You can't expect it to be installed on machines beyond your sphere of influence. An example are continuous integration server where Gradle is not installed and where you have no admin rights for the machine. Or what if you provide an open source project and you want to make it as easy as possible for your users to build it?

There is a simple and good news. Gradle provides a solution for this. It ships with a *Wrapper* task. [\[33\]](#) [\[34\]](#) You can create such a task in your build script.

Example 36.1. Wrapper task

build.gradle

```
task wrapper(type: Wrapper) {  
    gradleVersion = '0.6'  
}
```

The build master usually explicitly executes this task. After such an execution you find the following new or updated files in your project folder (in case the default configuration of the wrapper task is used).

```
project-root/  
  gradlew  
  gradlew.bat  
  gradle-wrapper.jar  
  gradle-wrapper.properties
```

All these files must be submitted to your version control system. The **gradlew** command can be used *exactly* the same way as the **gradle** command.

If you want to switch to a new version of Gradle you don't need to rerun the wrapper task. It is good enough to change the respective entry in the `gradle-wrapper.properties` file. But if there is for example an improvement in the gradle-wrapper functionality you need to regenerate the wrapper files.

36.1. Configuration

If you run Gradle with **gradlew**, Gradle checks if a Gradle distribution for the wrapper is available. If not it tries to download it, otherwise it delegates to the **gradle** command of this distribution with all the arguments passed originally to the **gradlew** command.

You can specify where the wrapper files should be stored (within your project directory):

Example 36.2. Configuration of wrapper task

build.gradle

```
task wrapper(type: Wrapper) {  
    gradleVersion = '0.6'  
    jarPath = 'wrapper'  
}
```

```
project-root/  
  gradlew  
  gradlew.bat  
  wrapper/  
    gradle-wrapper.jar  
    gradle-wrapper.properties
```

You can specify the download URL of the wrapper distribution. You can also specify where the wrapper distribution should be stored and unpacked (either within the project or within the gradle user home dir). If the wrapper is run and there is local archive of the wrapper distribution Gradle tries to download it and stores it at the specified place. If there is no unpacked wrapper distribution Gradle unpacks the local archive of the wrapper distribution at the specified place. All the configuration options have defaults except the version of the wrapper distribution.

For the details on how to configure the wrapper, see [Wrapper](#).

If you don't want any download to happen when your project is build via **gradlew**, simply add the Gradle distribution zip to your version control at the location specified by your wrapper configuration.

If you build via the wrapper, any existing Gradle distribution installed on the machine is ignored.

36.2. Unix file permissions

The Wrapper task adds appropriate file permissions to allow the execution for the gradlew *NIX command. Subversion preserves this file permission. We are not sure how other version control systems deal with this. What should always work is to execute `sh gradlew`.

36.3. Environment variable

Some rather exotic use cases might occur when working with the Gradle Wrapper. For example the continuous integration server goes down during unzipping the Gradle distribution. As the distribution directory exists **gradlew** delegates to it but the distribution is corrupt. Or the zip-distribution was not properly downloaded. When you have no admin right on the continuous integration server to remove the corrupt files, Gradle offers a solution via environment variables.

Table 36.1. Gradle wrapper environment variables

Variable Name	Meaning
GRADLE_WRAPPER_ALWAYS_UNPACK	If set to <code>true</code> , the distribution directory gets always deleted when gradlew is run and the distribution zip is freshly unpacked. If the zip is not there, Gradle tries to download it.
GRADLE_WRAPPER_ALWAYS_DOWNLOAD	If set to <code>true</code> , the distribution directory and the distribution zip gets always deleted when gradlew is run and the distribution zip is freshly downloaded.

[33] If you download the Gradle source distribution or check out Gradle from SVN, you can build Gradle via the Gradle wrapper.

[34] Gradle itself is continuously built by Bamboo and Teamcity via this wrapper. See <http://gradle.org/ci-server.html>

Embedding Gradle

t.b.d.

A

Potential Traps

A.1. Groovy script variables

For Gradle users it is important to understand how Groovy deals with script variables. Groovy has two types of script variables. One with a local scope and one with a script wide scope.

Example A.1. Variables scope: local and script wide

scope.groovy

```
String localScope1 = 'localScope1'
def localScope2 = 'localScope2'
scriptScope = 'scriptScope'

println localScope1
println localScope2
println scriptScope

closure = {
    println localScope1
    println localScope2
    println scriptScope
}

def method() {
    try {localScope1} catch(MissingPropertyException e) {println 'localScope1NotAvailable'}
    try {localScope2} catch(MissingPropertyException e) {println 'localScope2NotAvailable'}
    println scriptScope
}

closure.call()
method()
```

Output of gradle

```
> gradle
localScope1
localScope2
scriptScope
localScope1
localScope2
scriptScope
localScope1NotAvailable
localScope2NotAvailable
scriptScope
```

Variables which are declared with a type modifier are visible within closures but not visible within methods.

This is a heavily discussed behavior in the Groovy community. ^[35]

A.2. Configuration and execution phase

It is important to keep in mind that Gradle has a distinct configuration and execution phase (see [Chapter 30, The Build Lifecycle](#)).

Example A.2. Distinct configuration and execution phase

build.gradle

```
classesDir = new File('build/classes')
classesDir.mkdirs()
task clean << {
    ant.delete(dir: 'build')
}
task compile(dependsOn: 'clean') << {
    if (!classesDir.isDirectory()) {
        println 'The class directory does not exist. I can not operate'
        // do something
    }
    // do something
}
```

Output of `gradle -q compile`

```
> gradle -q compile
The class directory does not exist. I can not operate
```

As the creation of the directory happens during the configuration phase, the `clean` task removes the directory during the execution phase.

^[35] One of those discussions can be found here:
<http://www.nabble.com/script-scoping-question-td16034724.html>

Gradle Command Line

The **gradle** command has the following usage:

```
gradle [option...] [task-name...]
```

The command-line options available for the **gradle** command are listed below:

-?, -h, --help

Shows a help message.

-C, --cache

Specifies how compiled build scripts should be cached. Possible values are: `rebuild`, `off`, `on`. Default value is `on`. See [Section 11.4, “Caching”](#) for more details.

-D, --system-prop

Sets a system property of the JVM, for example `-Dmyprop=myvalue`.

-I, --init-script

Specifies an initialization script. See [Chapter 35, *Initialization Scripts*](#) for details.

-K, --default-import-file

Specifies the default import file.

-P, --project-prop

Sets a project property of the root project, for example `-Pmyprop=myvalue`.

-S, --full-stacktrace

Print out the full (very verbose) stacktrace for any exceptions. See [Chapter 15, *Logging*](#).

-a, --no-rebuild

Do not rebuild project dependencies.

-b, --build-file

Specifies the build file.

-c, --settings-file

Specifies the settings file.

-d, --debug

Log in debug mode (includes normal stacktrace). See [Chapter 15, *Logging*](#).

- e, --embedded**
Specify an embedded build script.
- g, --gradle-user-home**
Specifies the Gradle user home directory.
- gui**
Launches a GUI application. See [Chapter 10, Using the Gradle Graphical User Interface](#).
- i, --info**
Set log level to info. See [Chapter 15, Logging](#).
- l, --plugin-properties-file**
Specifies the plugin properties file.
- m, --dry-run**
Runs the build with all task actions disabled.
- n, --dependencies**
Show list of all project dependencies.
- no-imports**
Disable usage of default imports for build script files. See [Section C.3, "Using Gradle without IDE support"](#) for details.
- p, --project-dir**
Specifies the start directory for Gradle. Defaults to current directory.
- q, --quiet**
Log errors only. See [Chapter 15, Logging](#).
- r, --properties**
Show list of all available project properties.
- s, --stacktrace**
Print out the stacktrace also for user exceptions (e.g. compile error). See [Chapter 15, Logging](#).
- t, --tasks**
Show list of all available tasks and their dependencies.
- u, --no-search-upwards**
Don't search in parent folders for a `settings.gradle` file.
- v, --version**
Prints version info.
- x, --exclude-task**
Specifies a task to be excluded from execution.

The same information is printed to the console when you execute `gradle -h`.

Existing IDE Support and how to cope without it

C.1. IntelliJ

Gradle has been mainly developed with Idea IntelliJ and its very good Groovy plugin. Gradle's build script ^[36] has also been developed with the support of this IDE. IntelliJ allows you to define any filepattern to be interpreted as a Groovy script. In the case of Gradle you can define such a pattern for `build.gradle` and `settings.gradle`. This will already help very much. What is missing is the classpath to the Gradle binaries to offer content assistance for the Gradle classes. You might add the Gradle jar (which you can find in your distribution) to your project's classpath. It does not really belong there, but if you do this you have a fantastic IDE support for developing Gradle scripts. Of course if you use additional libraries for your build scripts they would further pollute your project classpath.

We hope that in the future `*.gradle` files get special treatment by IntelliJ and you will be able to define a specific classpath for them.

C.2. Eclipse

There is a Groovy plugin for eclipse. We don't know in what state it is and how it would support Gradle. In the next edition of this user guide we can hopefully write more about this.

C.3. Using Gradle without IDE support

What we can do for you is to spare you typing things like `throw new org.gradle.api.tasks.StopExecutionException()` and just type `throw new StopExecutionException()` instead. We do this by automatically adding a set of import statements to the Gradle scripts before Gradle executes them. This set is defined by a properties file `gradle-imports` in the Gradle distribution. It has the following content.

Figure C.1. gradle-imports

```
import org.gradle.*
import org.gradle.util.*
import org.gradle.api.*
import org.gradle.api.artifacts.*
import org.gradle.api.artifacts.dsl.*
import org.gradle.api.artifacts.specs.*
import org.gradle.api.dependencies.*
import org.gradle.api.execution.*
import org.gradle.api.file.*
import org.gradle.api.logging.*
import org.gradle.api.initialization.*
import org.gradle.api.invocation.*
import org.gradle.api.plugins.*
import org.gradle.api.plugins.quality.*
import org.gradle.api.specs.*
import org.gradle.api.tasks.*
import org.gradle.api.tasks.bundling.*
import org.gradle.api.tasks.compile.*
import org.gradle.api.tasks.javadoc.*
import org.gradle.api.tasks.testing.*
import org.gradle.api.tasks.util.*
import org.gradle.api.tasks.wrapper.*
```

You can define a project specific set of imports to be added to your build scripts. Just place a file called `grad` in your root project directory. If you start Gradle with the `{-I}` option, the imports defined in the Gradle distribution are disabled. The imports defined in your project directory are always used.

[36] Gradle is built with Gradle