

Gradle User Guide

Version 3.4.1

Copyright © 2007-2017 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

I. ABOUT GRADLE

1. Introduction
2. Overview

II. WORKING WITH EXISTING BUILDS

3. Installing Gradle
4. Using the Gradle Command-Line
5. The Gradle Wrapper
6. The Gradle Daemon
7. Dependency Management Basics
8. Introduction to multi-project builds
9. Continuous build
10. Composite builds
11. Using the Gradle Graphical User Interface
12. The Build Environment
13. Troubleshooting
14. Embedding Gradle using the Tooling API
15. Build Cache

III. WRITING GRADLE BUILD SCRIPTS

16. Build Script Basics
17. Build Init Plugin
18. Writing Build Scripts
19. More about Tasks
20. Working With Files
21. Using Ant from Gradle
22. The Build Lifecycle
23. Wrapper Plugin
24. Logging
25. Dependency Management
26. Multi-project Builds
27. Gradle Plugins
28. Standard Gradle plugins
29. The Project Report Plugin
30. The Build Dashboard Plugin
31. Comparing Builds
32. Publishing artifacts
33. The Maven Plugin
34. The Signing Plugin
35. Ivy Publishing (new)
36. Maven Publishing (new)
37. The Distribution Plugin
38. The Announce Plugin
39. The Build Announcements Plugin

IV. EXTENDING THE BUILD

40. Writing Custom Task Classes
41. Writing Custom Plugins
42. The Java Gradle Plugin Development Plugin
43. Organizing Build Logic
44. Initialization Scripts
45. The Gradle TestKit

V. BUILDING JVM PROJECTS

- 46. Java Quickstart
- 47. The Java Plugin
- 48. The Java Library Plugin
- 49. Web Application Quickstart
- 50. The War Plugin
- 51. The Ear Plugin
- 52. The Jetty Plugin
- 53. The Application Plugin
- 54. The Java Library Distribution Plugin
- 55. Groovy Quickstart
- 56. The Groovy Plugin
- 57. The Scala Plugin
- 58. The ANTLR Plugin
- 59. The Checkstyle Plugin
- 60. The CodeNarc Plugin
- 61. The FindBugs Plugin
- 62. The JDepend Plugin
- 63. The PMD Plugin
- 64. The JaCoCo Plugin
- 65. The OSGi Plugin
- 66. The Eclipse Plugins
- 67. The IDEA Plugin

VI. THE SOFTWARE MODEL

- 68. Rule based model configuration
- 69. Software model concepts
- 70. Implementing model rules in a plugin
- 71. Building Java Libraries
- 72. Building Play applications
- 73. Building native software
- 74. Extending the software model

VII. APPENDIX

- A. Gradle Samples
- B. Potential Traps
- C. The Feature Lifecycle
- D. Gradle Command Line
- E. Documentation licenses
- Glossary

List of Examples

- 4.1. Executing multiple tasks
- 4.2. Excluding tasks
- 4.3. Abbreviated task name
- 4.4. Abbreviated camel case task name
- 4.5. Selecting the project using a build file
- 4.6. Selecting the project using project directory
- 4.7. Forcing tasks to run
- 4.8. Obtaining information about projects
- 4.9. Providing a description for a project
- 4.10. Obtaining information about tasks

- 4.11. Changing the content of the task report
- 4.12. Obtaining more information about tasks
- 4.13. Obtaining detailed help for tasks
- 4.14. Obtaining information about dependencies
- 4.15. Filtering dependency report by configuration
- 4.16. Getting the insight into a particular dependency
- 4.17. Information about properties
- 5.1. Running the Wrapper task
- 5.2. Wrapper task
- 5.3. Wrapper generated files
- 5.4. Specifying the HTTP Basic Authentication credentials using system properties
- 5.5. Specifying the HTTP Basic Authentication credentials in `distributionUrl`
- 5.6. Generating a SHA-256 hash
- 5.7. Configuring SHA-256 checksum verification
- 7.1. Declaring dependencies
- 7.2. Definition of an external dependency
- 7.3. Shortcut definition of an external dependency
- 7.4. Usage of Maven central repository
- 7.5. Usage of JCenter repository
- 7.6. Usage of a remote Maven repository
- 7.7. Usage of a remote Ivy directory
- 7.8. Usage of a local Ivy directory
- 7.9. Publishing to an Ivy repository
- 7.10. Publishing to a Maven repository
- 8.1. Listing the projects in a build
- 10.1. Dependencies of my-app
- 10.2. Declaring a command-line composite
- 10.3. Declaring a separate composite
- 10.4. Depending on task from included build
- 10.5. Build that does not declare group attribute
- 10.6. Declaring the substitutions for an included build
- 10.7. Depending on a single task from an included build
- 10.8. Depending on a tasks with path in all included builds
- 11.1. Launching the GUI
- 12.1. Setting properties with a `gradle.properties` file
- 12.2. Configuring an HTTP proxy
- 12.3. Configuring an HTTPS proxy
- 14.1. Using the tooling API
- 16.1. Your first build script
- 16.2. Execution of a build script
- 16.3. A task definition shortcut
- 16.4. Using Groovy in Gradle's tasks
- 16.5. Using Groovy in Gradle's tasks
- 16.6. Declaration of task that depends on other task
- 16.7. Lazy dependsOn - the other task does not exist (yet)
- 16.8. Dynamic creation of a task
- 16.9. Accessing a task via API - adding a dependency
- 16.10. Accessing a task via API - adding behaviour

- 16.11. Accessing task as a property of the build script
- 16.12. Adding extra properties to a task
- 16.13. Using AntBuilder to execute ant.loadfile target
- 16.14. Using methods to organize your build logic
- 16.15. Defining a default task
- 16.16. Different outcomes of build depending on chosen tasks
- 18.1. Accessing property of the Project object
- 18.2. Using local variables
- 18.3. Using extra properties
- 18.4. Configuring arbitrary objects
- 18.5. Configuring arbitrary objects using a script
- 18.6. Groovy JDK methods
- 18.7. Property accessors
- 18.8. Method call without parentheses
- 18.9. List and map literals
- 18.10. Closure as method parameter
- 18.11. Closure delegates
- 19.1. Defining tasks
- 19.2. Defining tasks - using strings for task names
- 19.3. Defining tasks with alternative syntax
- 19.4. Accessing tasks as properties
- 19.5. Accessing tasks via tasks collection
- 19.6. Accessing tasks by path
- 19.7. Creating a copy task
- 19.8. Configuring a task - various ways
- 19.9. Configuring a task - with closure
- 19.10. Defining a task with closure
- 19.11. Adding dependency on task from another project
- 19.12. Adding dependency using task object
- 19.13. Adding dependency using closure
- 19.14. Adding a 'must run after' task ordering
- 19.15. Adding a 'should run after' task ordering
- 19.16. Task ordering does not imply task execution
- 19.17. A 'should run after' task ordering is ignored if it introduces an ordering cycle
- 19.18. Adding a description to a task
- 19.19. Overwriting a task
- 19.20. Skipping a task using a predicate
- 19.21. Skipping tasks with StopExecutionException
- 19.22. Enabling and disabling tasks
- 19.23. Custom task class
- 19.24. Ad-hoc task
- 19.25. Using runtime API with custom task type
- 19.26. Using skipWhenEmpty() via the runtime API
- 19.27. Inferred task dependency via task outputs
- 19.28. Inferred task dependency via a task argument
- 19.29. Declaring a method to add task inputs
- 19.30. Declaring a method to add a task as an input
- 19.31. Failed attempt at setting up an inferred task dependency

- 19.32. Setting up an inferred task dependency between output dir and input files
- 19.33. Setting up an inferred task dependency with files()
- 19.34. Setting up an inferred task dependency with builtBy()
- 19.35. Ignoring up-to-date checks
- 19.36. Task rule
- 19.37. Dependency on rule based tasks
- 19.38. Adding a task finalizer
- 19.39. Task finalizer for a failing task
- 20.1. Locating files
- 20.2. Creating a file collection
- 20.3. Using a file collection
- 20.4. Implementing a file collection
- 20.5. Creating a file tree
- 20.6. Using a file tree
- 20.7. Using an archive as a file tree
- 20.8. Specifying a set of files
- 20.9. Copying files using the copy task
- 20.10. Specifying copy task source files and destination directory
- 20.11. Selecting the files to copy
- 20.12. Copying files using the copy() method without up-to-date check
- 20.13. Copying files using the copy() method with up-to-date check
- 20.14. Renaming files as they are copied
- 20.15. Filtering files as they are copied
- 20.16. Nested copy specs
- 20.17. Using the Sync task to copy dependencies
- 20.18. Creating a ZIP archive
- 20.19. Creation of ZIP archive
- 20.20. Configuration of archive task - custom archive name
- 20.21. Configuration of archive task - appendix & classifier
- 20.22. Activating reproducible archives
- 21.1. Using an Ant task
- 21.2. Passing nested text to an Ant task
- 21.3. Passing nested elements to an Ant task
- 21.4. Using an Ant type
- 21.5. Using a custom Ant task
- 21.6. Declaring the classpath for a custom Ant task
- 21.7. Using a custom Ant task and dependency management together
- 21.8. Importing an Ant build
- 21.9. Task that depends on Ant target
- 21.10. Adding behaviour to an Ant target
- 21.11. Ant target that depends on Gradle task
- 21.12. Renaming imported Ant targets
- 21.13. Setting an Ant property
- 21.14. Getting an Ant property
- 21.15. Setting an Ant reference
- 21.16. Getting an Ant reference
- 21.17. Fine tuning Ant logging
- 22.1. Single project build

- 22.2. Hierarchical layout
- 22.3. Flat layout
- 22.4. Modification of elements of the project tree
- 22.5. Adding of test task to each project which has certain property set
- 22.6. Notifications
- 22.7. Setting of certain property to all tasks
- 22.8. Logging of start and end of each task execution
- 24.1. Using stdout to write log messages
- 24.2. Writing your own log messages
- 24.3. Using SLF4J to write log messages
- 24.4. Configuring standard output capture
- 24.5. Configuring standard output capture for a task
- 24.6. Customizing what Gradle logs
- 25.1. Definition of a configuration
- 25.2. Accessing a configuration
- 25.3. Configuration of a configuration
- 25.4. Module dependencies
- 25.5. Artifact only notation
- 25.6. Dependency with classifier
- 25.7. Iterating over a configuration
- 25.8. Client module dependencies - transitive dependencies
- 25.9. Project dependencies
- 25.10. File dependencies
- 25.11. Generated file dependencies
- 25.12. Gradle API dependencies
- 25.13. Gradle's Groovy dependencies
- 25.14. Excluding transitive dependencies
- 25.15. Optional attributes of dependencies
- 25.16. Collections and arrays of dependencies
- 25.17. Dependency configurations
- 25.18. Dependency configurations for project
- 25.19. Configuration.copy
- 25.20. Accessing declared dependencies
- 25.21. Configuration.files
- 25.22. Configuration.files with spec
- 25.23. Configuration.copy
- 25.24. Configuration.copy vs. Configuration.files
- 25.25. Adding central Maven repository
- 25.26. Adding Bintray's JCenter Maven repository
- 25.27. Using Bintray's JCenter with HTTP
- 25.28. Adding the local Maven cache as a repository
- 25.29. Adding custom Maven repository
- 25.30. Adding additional Maven repositories for JAR files
- 25.31. Accessing password protected Maven repository
- 25.32. Flat repository resolver
- 25.33. Ivy repository
- 25.34. Ivy repository with named layout
- 25.35. Ivy repository with pattern layout

- 25.36. Ivy repository with multiple custom patterns
- 25.37. Ivy repository with Maven compatible layout
- 25.38. Ivy repository
- 25.39. Declaring a Maven and Ivy repository
- 25.40. Providing credentials to a Maven and Ivy repository
- 25.41. Declaring a S3 backed Maven and Ivy repository
- 25.42. Declaring a S3 backed Maven and Ivy repository using IAM
- 25.43. Configure repository to use only digest authentication
- 25.44. Configure repository to use preemptive authentication
- 25.45. Accessing a repository
- 25.46. Configuration of a repository
- 25.47. Definition of a custom repository
- 25.48. Forcing consistent version for a group of libraries
- 25.49. Using a custom versioning scheme
- 25.50. Blacklisting a version with a replacement
- 25.51. Changing dependency group and/or name at the resolution
- 25.52. Substituting a module with a project
- 25.53. Substituting a project with a module
- 25.54. Conditionally substituting a dependency
- 25.55. Specifying default dependencies on a configuration
- 25.56. Enabling dynamic resolve mode
- 25.57. 'Latest' version selector
- 25.58. Custom status scheme
- 25.59. Custom status scheme by module
- 25.60. Ivy component metadata rule
- 25.61. Rule source component metadata rule
- 25.62. Component selection rule
- 25.63. Component selection rule with module target
- 25.64. Component selection rule with metadata
- 25.65. Component selection rule using a rule source object
- 25.66. Declaring module replacement
- 25.67. Dynamic version cache control
- 25.68. Changing module cache control
- 26.1. Multi-project tree - water & bluewhale projects
- 26.2. Build script of water (parent) project
- 26.3. Multi-project tree - water, bluewhale & krill projects
- 26.4. Water project build script
- 26.5. Defining common behavior of all projects and subprojects
- 26.6. Defining specific behaviour for particular project
- 26.7. Defining specific behaviour for project krill
- 26.8. Adding custom behaviour to some projects (filtered by project name)
- 26.9. Adding custom behaviour to some projects (filtered by project properties)
- 26.10. Running build from subproject
- 26.11. Evaluation and execution of projects
- 26.12. Evaluation and execution of projects
- 26.13. Running tasks by their absolute path
- 26.14. Dependencies and execution order
- 26.15. Dependencies and execution order

- 26.16. Dependencies and execution order
- 26.17. Declaring dependencies
- 26.18. Declaring dependencies
- 26.19. Cross project task dependencies
- 26.20. Configuration time dependencies
- 26.21. Configuration time dependencies - evaluationDependsOn
- 26.22. Configuration time dependencies
- 26.23. Dependencies - real life example - crossproject configuration
- 26.24. Project lib dependencies
- 26.25. Project lib dependencies
- 26.26. Fine grained control over dependencies
- 26.27. Build and Test Single Project
- 26.28. Partial Build and Test Single Project
- 26.29. Build and Test Depended On Projects
- 26.30. Build and Test Dependent Projects
- 27.1. Applying a script plugin
- 27.2. Applying a core plugin
- 27.3. Applying a community plugin
- 27.4. Applying plugins only on certain subprojects.
- 27.5. Using plugins from custom plugin repositories.
- 27.6. Complete Plugin Publishing Sample
- 27.7. Applying a binary plugin
- 27.8. Applying a binary plugin by type
- 27.9. Applying a plugin with the buildscript block
- 30.1. Using the Build Dashboard plugin
- 32.1. Defining an artifact using an archive task
- 32.2. Defining an artifact using a file
- 32.3. Customizing an artifact
- 32.4. Map syntax for defining an artifact using a file
- 32.5. Configuration of the upload task
- 33.1. Using the Maven plugin
- 33.2. Creating a stand alone pom.
- 33.3. Upload of file to remote Maven repository
- 33.4. Upload of file via SSH
- 33.5. Customization of pom
- 33.6. Builder style customization of pom
- 33.7. Modifying auto-generated content
- 33.8. Customization of Maven installer
- 33.9. Generation of multiple poms
- 33.10. Accessing a mapping configuration
- 34.1. Using the Signing plugin
- 34.2. Signing a configuration
- 34.3. Signing a configuration output
- 34.4. Signing a task
- 34.5. Signing a task output
- 34.6. Conditional signing
- 34.7. Signing a POM for deployment
- 35.1. Applying the “ivy-publish” plugin

- 35.2. Publishing a Java module to Ivy
- 35.3. Publishing additional artifact to Ivy
- 35.4. customizing the publication identity
- 35.5. Customizing the module descriptor file
- 35.6. Publishing multiple modules from a single project
- 35.7. Declaring repositories to publish to
- 35.8. Choosing a particular publication to publish
- 35.9. Publishing all publications via the “publish” lifecycle task
- 35.10. Generating the Ivy module descriptor file
- 35.11. Publishing a Java module
- 35.12. Example generated ivy.xml
- 36.1. Applying the 'maven-publish' plugin
- 36.2. Adding a MavenPublication for a Java component
- 36.3. Adding additional artifact to a MavenPublication
- 36.4. customizing the publication identity
- 36.5. Modifying the POM file
- 36.6. Publishing multiple modules from a single project
- 36.7. Declaring repositories to publish to
- 36.8. Publishing a project to a Maven repository
- 36.9. Publish a project to the Maven local repository
- 36.10. Generate a POM file without publishing
- 37.1. Using the distribution plugin
- 37.2. Adding extra distributions
- 37.3. Configuring the main distribution
- 37.4. publish main distribution
- 38.1. Using the announce plugin
- 38.2. Configure the announce plugin
- 38.3. Using the announce plugin
- 39.1. Using the build announcements plugin
- 39.2. Using the build announcements plugin from an init script
- 40.1. Defining a custom task
- 40.2. A hello world task
- 40.3. A customizable hello world task
- 40.4. A build for a custom task
- 40.5. A custom task
- 40.6. Using a custom task in another project
- 40.7. Testing a custom task
- 40.8. Defining an incremental task action
- 40.9. Running the incremental task for the first time
- 40.10. Running the incremental task with unchanged inputs
- 40.11. Running the incremental task with updated input files
- 40.12. Running the incremental task with an input file removed
- 40.13. Running the incremental task with an output file removed
- 40.14. Running the incremental task with an input property changed
- 41.1. A custom plugin
- 41.2. A custom plugin extension
- 41.3. A custom plugin with configuration closure
- 41.4. Evaluating file properties lazily

- 41.5. A build for a custom plugin
- 41.6. Wiring for a custom plugin
- 41.7. Using a custom plugin in another project
- 41.8. Applying a community plugin with the plugins DSL
- 41.9. Testing a custom plugin
- 41.10. Using the Java Gradle Plugin Development plugin
- 41.11. Managing domain objects
- 42.1. Using the Java Gradle Plugin Development plugin
- 42.2. Using the gradlePlugin { } block.
- 43.1. Using inherited properties and methods
- 43.2. Using injected properties and methods
- 43.3. Configuring the project using an external build script
- 43.4. Custom buildSrc build script
- 43.5. Adding subprojects to the root buildSrc project
- 43.6. Running another build from a build
- 43.7. Declaring external dependencies for the build script
- 43.8. A build script with external dependencies
- 43.9. Ant optional dependencies
- 44.1. Using init script to perform extra configuration before projects are evaluated
- 44.2. Declaring external dependencies for an init script
- 44.3. An init script with external dependencies
- 44.4. Using plugins in init scripts
- 45.1. Declaring the TestKit dependency
- 45.2. Declaring the JUnit dependency
- 45.3. Using GradleRunner with JUnit
- 45.4. Using GradleRunner with Spock
- 45.5. Making the code under test classpath available to the tests
- 45.6. Injecting the code under test classes into test builds
- 45.7. Using the Java Gradle Development plugin for generating the plugin metadata
- 45.8. Automatically injecting the code under test classes into test builds
- 45.9. Reconfiguring the classpath generation conventions of the Java Gradle Development plugin
- 45.10. Specifying a Gradle version for test execution
- 46.1. Using the Java plugin
- 46.2. Building a Java project
- 46.3. Adding Maven repository
- 46.4. Adding dependencies
- 46.5. Customization of MANIFEST.MF
- 46.6. Adding a test system property
- 46.7. Publishing the JAR file
- 46.8. Eclipse plugin
- 46.9. Java example - complete build file
- 46.10. Multi-project build - hierarchical layout
- 46.11. Multi-project build - settings.gradle file
- 46.12. Multi-project build - common configuration
- 46.13. Multi-project build - dependencies between projects
- 46.14. Multi-project build - distribution file
- 47.1. Using the Java plugin
- 47.2. Custom Java source layout

- 47.3. Accessing a source set
- 47.4. Configuring the source directories of a source set
- 47.5. Defining a source set
- 47.6. Defining source set dependencies
- 47.7. Compiling a source set
- 47.8. Assembling a JAR for a source set
- 47.9. Generating the Javadoc for a source set
- 47.10. Running tests in a source set
- 47.11. Declaring annotation processors
- 47.12. Filtering tests in the build script
- 47.13. JUnit Categories
- 47.14. Grouping TestNG tests
- 47.15. Preserving order of TestNG tests
- 47.16. Grouping TestNG tests by instances
- 47.17. Creating a unit test report for subprojects
- 47.18. Customization of MANIFEST.MF
- 47.19. Creating a manifest object.
- 47.20. Separate MANIFEST.MF for a particular archive
- 47.21. Configure Java 6 build
- 48.1. Using the Java Library plugin
- 48.2. Declaring API and implementation dependencies
- 48.3. Making the difference between API and implementation
- 48.4. Declaring API and implementation dependencies
- 48.5. Configuring the Groovy plugin to work with Java Library
- 49.1. War plugin
- 49.2. Running web application with Jetty plugin
- 50.1. Using the War plugin
- 50.2. Customization of war plugin
- 51.1. Using the Ear plugin
- 51.2. Customization of ear plugin
- 52.1. Using the Jetty plugin
- 53.1. Using the application plugin
- 53.2. Configure the application main class
- 53.3. Configure default JVM settings
- 53.4. Include output from other tasks in the application distribution
- 53.5. Automatically creating files for distribution
- 54.1. Using the Java library distribution plugin
- 54.2. Configure the distribution name
- 54.3. Include files in the distribution
- 55.1. Groovy plugin
- 55.2. Dependency on Groovy
- 55.3. Groovy example - complete build file
- 56.1. Using the Groovy plugin
- 56.2. Custom Groovy source layout
- 56.3. Configuration of Groovy dependency
- 56.4. Configuration of Groovy test dependency
- 56.5. Configuration of bundled Groovy dependency
- 56.6. Configuration of Groovy file dependency

- 56.7. Configure Java 6 build for Groovy
- 57.1. Using the Scala plugin
- 57.2. Custom Scala source layout
- 57.3. Declaring a Scala dependency for production code
- 57.4. Declaring a Scala dependency for test code
- 57.5. Declaring a version of the Zinc compiler to use
- 57.6. Forcing a scala-library dependency for all configurations
- 57.7. Forcing a scala-library dependency for the zinc configuration
- 57.8. Adjusting memory settings
- 57.9. Forcing all code to be compiled
- 57.10. Configure Java 6 build for Scala
- 57.11. Explicitly specify a target IntelliJ IDEA version
- 58.1. Using the ANTLR plugin
- 58.2. Declare ANTLR version
- 58.3. setting custom max heap size and extra arguments for ANTLR
- 59.1. Using the Checkstyle plugin
- 59.2. Customizing the HTML report
- 60.1. Using the CodeNarc plugin
- 61.1. Using the FindBugs plugin
- 61.2. Customizing the HTML report
- 62.1. Using the JDepend plugin
- 63.1. Using the PMD plugin
- 64.1. Applying the JaCoCo plugin
- 64.2. Configuring JaCoCo plugin settings
- 64.3. Configuring test task
- 64.4. Configuring violation rules
- 64.5. Configuring test task
- 64.6. Using application plugin to generate code coverage data
- 64.7. Coverage reports generated by applicationCodeCoverageReport
- 65.1. Using the OSGi plugin
- 65.2. Configuration of OSGi MANIFEST.MF file
- 66.1. Using the Eclipse plugin
- 66.2. Using the Eclipse WTP plugin
- 66.3. Partial Overwrite for Classpath
- 66.4. Partial Overwrite for Project
- 66.5. Export Dependencies
- 66.6. Customizing the XML
- 67.1. Using the IDEA plugin
- 67.2. Partial Rewrite for Module
- 67.3. Partial Rewrite for Project
- 67.4. Export Dependencies
- 67.5. Customizing the XML
- 68.1. applying a rule source plugin
- 68.2. a model creation rule
- 68.3. a model mutation rule
- 68.4. creating a task
- 68.5. a managed type
- 68.6. a String property

- 68.7. a File property
- 68.8. a Long property
- 68.9. a boolean property
- 68.10. an int property
- 68.11. a managed property
- 68.12. an enumeration type property
- 68.13. a managed set
- 68.14. strongly modelling sources sets
- 68.15. a DSL example applying a rule to every element in a scope
- 68.16. DSL configuration rule
- 68.17. Configuration run when required
- 68.18. Configuration not run when not required
- 68.19. DSL creation rule
- 68.20. DSL creation rule without initialization
- 68.21. Initialization before configuration
- 68.22. Nested DSL creation rule
- 68.23. Nested DSL configuration rule
- 68.24. DSL configuration rule for each element in a map
- 68.25. Nested DSL property configuration
- 68.26. a DSL example showing type conversions
- 68.27. a DSL rule using inputs
- 68.28. model task output
- 71.1. Using the Java software plugins
- 71.2. Creating a java library
- 71.3. Configuring a source set
- 71.4. Creating a new source set
- 71.5. The components report
- 71.6. Declaring a dependency onto a library
- 71.7. Declaring a dependency onto a project with an explicit library
- 71.8. Declaring a dependency onto a project with an implicit library
- 71.9. Declaring a dependency onto a library published to a Maven repository
- 71.10. Declaring a module dependency using shorthand notation
- 71.11. Configuring repositories for dependency resolution
- 71.12. Specifying api packages
- 71.13. Specifying api dependencies
- 71.14. Main sources
- 71.15. Client component
- 71.16. Broken client component
- 71.17. Recompiling the client
- 71.18. Declaring target platforms
- 71.19. Declaring binary specific sources
- 71.20. Declaring target platforms
- 71.21. Using the JUnit plugin
- 71.22. Executing the test suite
- 71.23. Executing the test suite
- 71.24. Declaring a component under test
- 71.25. Declaring local Java installations
- 72.1. Using the Play plugin

- 72.2. The components report
- 72.3. Selecting a version of the Play Framework
- 72.4. Adding dependencies to a Play application
- 72.5. Adding extra source sets to a Play application
- 72.6. Configuring Scala compiler options
- 72.7. Configuring routes style
- 72.8. Configuring a custom asset pipeline
- 72.9. Configuring dependencies on Play subprojects
- 72.10. Add extra files to a Play application distribution
- 72.11. Applying both the Play and IDEA plugins
- 73.1. Defining a library component
- 73.2. Defining executable components
- 73.3. Sample build
- 73.4. Dependent components report
- 73.5. Dependent components report
- 73.6. Report of components that depends on the operators component
- 73.7. Report of components that depends on the operators component, including test suites
- 73.8. Assemble components that depends on the passing/static binary of the operators component
- 73.9. Build components that depends on the passing/static binary of the operators component
- 73.10. Adding a custom check task
- 73.11. Running checks for a given binary
- 73.12. The components report
- 73.13. The 'cpp' plugin
- 73.14. C++ source set
- 73.15. The 'c' plugin
- 73.16. C source set
- 73.17. The 'assembler' plugin
- 73.18. The 'objective-c' plugin
- 73.19. The 'objective-cpp' plugin
- 73.20. Settings that apply to all binaries
- 73.21. Settings that apply to all shared libraries
- 73.22. Settings that apply to all binaries produced for the 'main' executable component
- 73.23. Settings that apply only to shared libraries produced for the 'main' library component
- 73.24. The 'windows-resources' plugin
- 73.25. Configuring the location of Windows resource sources
- 73.26. Building a resource-only dll
- 73.27. Providing a library dependency to the source set
- 73.28. Providing a library dependency to the binary
- 73.29. Declaring project dependencies
- 73.30. Creating a precompiled header file
- 73.31. Including a precompiled header file in a source file
- 73.32. Configuring a precompiled header
- 73.33. Defining build types
- 73.34. Configuring debug binaries
- 73.35. Defining platforms
- 73.36. Defining flavors
- 73.37. Targeting a component at particular platforms
- 73.38. Building all possible variants

- 73.39. Defining tool chains
- 73.40. Reconfigure tool arguments
- 73.41. Defining target platforms
- 73.42. Registering CUnit tests
- 73.43. Running CUnit tests
- 73.44. Registering GoogleTest tests
- 74.1. an example of using a custom software model
- 74.2. Declare a custom component
- 74.3. Register a custom component
- 74.4. Declare a custom binary
- 74.5. Register a custom binary
- 74.6. Declare a custom source set
- 74.7. Register a custom source set
- 74.8. Generates documentation binaries
- 74.9. Generates tasks for text source sets
- 74.10. Register a custom source set
- 74.11. an example of using a custom software model
- 74.12. components report
- 74.13. public type and internal view declaration
- 74.14. type registration
- 74.15. public and internal data mutation
- 74.16. example build script and model report output
- B.1. Variables scope: local and script wide
- B.2. Distinct configuration and execution phase

Part I. About Gradle

1

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.

Throughout the user guide, you will find some diagrams that represent dependency relationships between Gradle tasks. These use something analogous to the UML dependency notation, which renders an arrow from one task to the task that the first task depends on.

2.1. Features

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

Declarative builds and build-by-convention

At the heart of Gradle lies a rich extensible Domain Specific Language (DSL) based on Groovy. Gradle pushes declarative builds to the next level by providing declarative language elements that you can assemble as you like. Those elements also provide build-by-convention support for Java, Groovy, OSGi, Web and Scala projects. Even more, this declarative language is extensible. Add your own new language elements or enhance the existing ones, thus providing concise, maintainable and comprehensible builds.

Language for dependency based programming

The declarative language lies on top of a general purpose task graph, which you can fully leverage in your builds. It provides utmost flexibility to adapt Gradle to your unique needs.

Structure your build

The suppleness and richness of Gradle finally allows you to apply common design principles to your build. For example, it is very easy to compose your build from reusable pieces of build logic. Inline stuff where unnecessary indirections would be inappropriate. Don't be forced to tear apart what belongs together (e.g. in your project hierarchy). Avoid smells like shotgun changes or divergent change that turn your build into a maintenance nightmare. At last you can create a well structured, easily maintained, comprehensible build.

Deep API

From being a pleasure to be used embedded to its many hooks over the whole lifecycle of build execution, Gradle allows you to monitor and customize its configuration and execution behavior to its very core.

Gradle scales

Gradle scales very well. It significantly increases your productivity, from simple single project builds up to huge enterprise multi-project builds. This is true for structuring the build. With the state-of-art incremental build function, this is also true for tackling the performance pain many large enterprise builds suffer from.

Multi-project builds

Gradle's support for multi-project build is outstanding. Project dependencies are first class citizens. 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.

Gradle provides partial builds. If you build a single subproject Gradle takes care of building all the subprojects that subproject depends on. You can also choose to rebuild the subprojects that depend on a particular subproject. Together with incremental builds this is a big time saver for larger builds.

Many ways to manage your dependencies

Different teams prefer different ways to manage their external dependencies. Gradle provides convenient support for any strategy. From transitive dependency management with remote Maven and Ivy repositories to jars or directories on the local file system.

Gradle is the first build integration tool

Ant tasks are first class citizens. Even more interesting, Ant projects are first class citizens as well. Gradle provides a deep import for any Ant project, turning Ant targets into native Gradle tasks at runtime. You can depend on them from Gradle, you can enhance them from Gradle, you can even declare dependencies on Gradle tasks in your build.xml. The same integration is provided for properties, paths, etc ...

Gradle fully supports your existing Maven or Ivy repository infrastructure for publishing and retrieving dependencies. Gradle also provides a converter for turning a Maven pom.xml into a Gradle script. Runtime imports of Maven projects will come soon.

Ease of migration

Gradle can adapt to any structure you have. Therefore you can always develop your Gradle build in the same branch where your production build lives and both can evolve in parallel. We usually recommend to write tests that make sure that the produced artifacts are similar. That way migration is as less disruptive and as reliable as possible. This is following the best-practices for refactoring by applying baby steps.

Groovy

Gradle's build scripts are written in Groovy, not XML. But unlike other approaches this is not for simply exposing the raw scripting power of a dynamic language. That would just lead to a very difficult to maintain build. The whole design of Gradle is oriented towards being used as a language, not as a rigid framework. And Groovy is our glue that allows you to tell your individual story with the abstractions Gradle (or you) provide. Gradle provides some standard stories but they are not privileged in any form. This is for us a major distinguishing feature compared to other declarative build systems. Our Groovy support is not just sugar coating. The whole Gradle API is fully Groovy-ized. Adding Groovy results in an enjoyable and productive experience.

The Gradle wrapper

The Gradle Wrapper allows you to execute Gradle builds on machines where Gradle is not installed. This is useful for example for some continuous integration servers. It is also useful for an open source project to keep the barrier low for building it. The wrapper is also very interesting for the enterprise. It is a zero administration approach for the client machines. It also enforces the usage of a particular Gradle version thus minimizing support issues.

Free and open source

Gradle is an open source project, and is licensed under the ASL.

2.2. Why Groovy?

We think the advantages of an internal DSL (based on a dynamic language) over XML are tremendous when used in *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. In such projects the team members will be very familiar with Java. We think a build should be as transparent as possible to *all* team members.

In that case, you might argue why we don't just use Java 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 because of the limitations of Java, such a build language would not be as nice, expressive and powerful as it could be. ^[1] Languages like Python, Groovy or Ruby do a much better job here. We have chosen Groovy as it offers by far the greatest transparency for Java people. Its base syntax is the same as Java's as well as its type system, its package structure and other things. Groovy provides much more on top of that, but with the common foundation of Java.

For Java developers with Python or Ruby knowledge or the desire to learn them, the above arguments don't apply. The Gradle design is well-suited for creating another build script engine in JRuby or Jython. It just doesn't have the highest priority for us at the moment. We happily support any community effort to create additional build script engines.

[1] 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.

Part II. Working with existing builds

Installing Gradle

3.1. Prerequisites

Gradle requires a Java JDK or JRE to be installed, version 7 or higher (to check, use `java -version`). Gradle ships with its own Groovy library, therefore Groovy does not need to be installed. Any existing Groovy installation is ignored by Gradle.

Gradle uses whatever JDK it finds in your path. Alternatively, you can set the `JAVA_HOME` environment variable to point to the installation 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 DSL reference guide.
- The API documentation (Javadoc).
- Extensive samples, including the examples referenced in the user guide, along with some complete and more complex builds you can use as a starting point for your own build.
- The binary sources. This is for reference only. If you want to build Gradle you need to download the source distribution or checkout the sources from the source repository. See the Gradle web site for details.

3.4. Environment variables

For running Gradle, firstly add the environment variable `GRADLE_HOME`. This should point to the unpacked files from the Gradle website. Next add `GRADLE_HOME/bin` to your `PATH` environment variable. Usually, this is sufficient to run Gradle.

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**. The output shows the Gradle version and also the local environment configuration (Groovy, JVM version, OS, etc.). The displayed Gradle version should match the distribution you have downloaded.

3.6. JVM options

JVM options for running Gradle can be set via environment variables. You can use either `GRADLE_OPTS` or `JAVA_OPTS`, or both. `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.

Note that it's not currently possible to set JVM options for Gradle on the command line.

Using the Gradle Command-Line

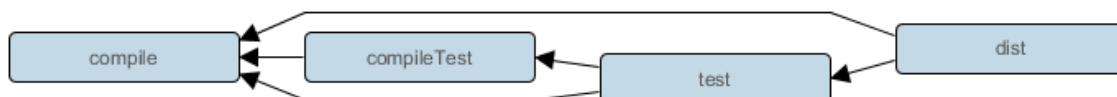
This chapter introduces the basics of the Gradle command-line. You run a build using the **gradle** command, which you have already seen in action in previous chapters.

4.1. 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 how it came to be included in the build: whether it was specified on the command-line, or as 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 test** for this build script results in the `compile` task being executed only once.

Figure 4.1. Task dependencies



Example 4.1. Executing multiple tasks

build.gradle

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

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

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

task dist(dependsOn: [compile, test]) {
    doLast {
        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
```

Each task is executed only once, so **gradle test test** is exactly the same as **gradle test**.

4.2. 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 4.2. 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.

4.3. Continuing the build when a failure occurs

By default, Gradle will abort execution and fail the build as soon as any task fails. This allows the build to complete sooner, but hides other failures that would have occurred. In order to discover as many failures as possible in a single build execution, you can use the `--continue` option.

When executed with `--continue`, Gradle will execute *every* task to be executed where all of the dependencies for that task completed without failure, instead of stopping as soon as the first failure is encountered. Each of the encountered failures will be reported at the end of the build.

If a task fails, any subsequent tasks that were depending on it will not be executed, as it is not safe to do so. For example, tests will not run if there is a compilation failure in the code under test; because the `test` task will depend on the compilation task (either directly or indirectly).

4.4. 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 4.3. Abbreviated task name

Output of **gradle di**

```
> gradle di
: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 `compileTest` by running **gradle compTest** or even **gradle cT**

Example 4.4. 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.

4.5. 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. If you use `-b` option then `settings.gradle` file is not used. Example:

Example 4.5. Selecting the project using a build file

subdir/myproject.gradle

```
task hello {
    doLast {
        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. For multi-project builds you should use `-p` option instead of `-b` option.

Example 4.6. 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'.
```

4.6. Forcing tasks to execute

Many tasks, particularly those provided by Gradle itself, support incremental builds. Such tasks can determine whether they need to run or not based on whether their inputs or outputs have changed since the last time they ran. You can easily identify tasks that take part in incremental build when Gradle displays the text `UP-TO-DATE` next to their name during a build run.

You may on occasion want to force Gradle to run all the tasks, ignoring any up-to-date checks. If that's the case, simply use the `--rerun-tasks` option. Here's the output when running a task both without and with `--rerun-tasks`:

Example 4.7. Forcing tasks to run

Output of **gradle doIt**

```
> gradle doIt
:doIt UP-TO-DATE
```

Output of **gradle --rerun-tasks doIt**

```
> gradle --rerun-tasks doIt
:doIt
```

Note that this will force all required tasks to execute, not just the ones you specify on the command line. It's a little like running a `clean`, but without the build's generated output being deleted.

4.7. Obtaining information about your build

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

In addition to the built-in tasks shown below, you can also use the project report plugin to add tasks to your project which will generate these reports.

4.7.1. Listing projects

Running **gradle projects** gives you a list of the sub-projects of the selected project, displayed in a hierarchy. Here is an example:

Example 4.8. Obtaining information about projects

Output of **gradle -q projects**

```
> gradle -q projects

-----
Root project
-----

Root project 'projectReports'
+--- Project ':api' - The shared API for the application
\--- Project ':webapp' - The Web application implementation

To see a list of the tasks of a project, run gradle <project-path>:tasks
For example, try running gradle :api:tasks
```

The report shows the description of each project, if specified. You can provide a description for a project by setting the `description` property:

Example 4.9. Providing a description for a project

build.gradle

```
description = 'The shared API for the application'
```

4.7.2. Listing tasks

Running **gradle tasks** gives you a list of the main tasks of the selected project. This report shows the default tasks for the project, if any, and a description for each task. Below is an example of this report:

Example 4.10. Obtaining information about tasks

Output of `gradle -q tasks`

```
> gradle -q tasks

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

Default tasks: dists

Build tasks
-----
clean - Deletes the build directory (build)
dists - Builds the distribution
libs - Builds the JAR

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

Help tasks
-----
buildEnvironment - Displays all buildscript dependencies declared in root project 'p
components - Displays the components produced by root project 'projectReports'. [inc
dependencies - Displays all dependencies declared in root project 'projectReports'.
dependencyInsight - Displays the insight into a specific dependency in root project
dependentComponents - Displays the dependent components of components in root projec
help - Displays a help message.
model - Displays the configuration model of root project 'projectReports'. [incubati
projects - Displays the sub-projects of root project 'projectReports'.
properties - Displays the properties of root project 'projectReports'.
tasks - Displays the tasks runnable from root project 'projectReports' (some of the

To see all tasks and more detail, run gradle tasks --all

To see more detail about a task, run gradle help --task <task>
```

By default, this report shows only those tasks which have been assigned to a task group, so-called *visible* tasks. You can do this by setting the `group` property for the task. You can also set the `description` property, to provide a description to be included in the report.

Example 4.11. Changing the content of the task report

`build.gradle`

```
dists {
    description = 'Builds the distribution'
    group = 'build'
}
```

You can obtain more information in the task listing using the `--all` option. With this option, the task report lists all tasks in the project, including tasks which have not been assigned to a task group, so-called *hidden* tasks. Here is an example:

Example 4.12. Obtaining more information about tasks

Output of `gradle -q tasks --all`


```
> gradle -q tasks --all
```

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

Default tasks: dists

Build tasks

```
-----  
clean - Deletes the build directory (build)  
api:clean - Deletes the build directory (build)  
webapp:clean - Deletes the build directory (build)  
dists - Builds the distribution  
api:libs - Builds the JAR  
webapp:libs - Builds the JAR
```

Build Setup tasks

```
-----  
init - Initializes a new Gradle build. [incubating]  
wrapper - Generates Gradle wrapper files. [incubating]
```

Help tasks

```
-----  
buildEnvironment - Displays all buildscript dependencies declared in root project 'p  
api:buildEnvironment - Displays all buildscript dependencies declared in project ':a  
webapp:buildEnvironment - Displays all buildscript dependencies declared in project  
components - Displays the components produced by root project 'projectReports'. [inc  
api:components - Displays the components produced by project ':api'. [incubating]  
webapp:components - Displays the components produced by project ':webapp'. [incubati  
dependencies - Displays all dependencies declared in root project 'projectReports'.  
api:dependencies - Displays all dependencies declared in project ':api'.  
webapp:dependencies - Displays all dependencies declared in project ':webapp'.  
dependencyInsight - Displays the insight into a specific dependency in root project  
api:dependencyInsight - Displays the insight into a specific dependency in project '  
webapp:dependencyInsight - Displays the insight into a specific dependency in projec  
dependentComponents - Displays the dependent components of components in root projec  
api:dependentComponents - Displays the dependent components of components in project  
webapp:dependentComponents - Displays the dependent components of components in proj  
help - Displays a help message.  
api:help - Displays a help message.  
webapp:help - Displays a help message.  
model - Displays the configuration model of root project 'projectReports'. [incubati  
api:model - Displays the configuration model of project ':api'. [incubating]  
webapp:model - Displays the configuration model of project ':webapp'. [incubating]  
projects - Displays the sub-projects of root project 'projectReports'.  
api:projects - Displays the sub-projects of project ':api'.  
webapp:projects - Displays the sub-projects of project ':webapp'.  
properties - Displays the properties of root project 'projectReports'.  
api:properties - Displays the properties of project ':api'.  
webapp:properties - Displays the properties of project ':webapp'.  
tasks - Displays the tasks runnable from root project 'projectReports' (some of the  
api:tasks - Displays the tasks runnable from project ':api'.  
webapp:tasks - Displays the tasks runnable from project ':webapp'.
```

Other tasks

```
-----  
api:compile - Compiles the source files  
webapp:compile - Compiles the source files  
docs - Builds the documentation
```

4.7.3. Show task usage details

Running **gradle help --task someTask** gives you detailed information about a specific task or multiple tasks matching the given task name in your multi-project build. Below is an example of this detailed information:

Example 4.13. Obtaining detailed help for tasks

Output of **gradle -q help --task libs**

```
> gradle -q help --task libs
Detailed task information for libs

Paths
    :api:libs
    :webapp:libs

Type
    Task (org.gradle.api.Task)

Description
    Builds the JAR

Group
    build
```

This information includes the full task path, the task type, possible commandline options and the description of the given task.

4.7.4. Listing project dependencies

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

Example 4.14. Obtaining information about dependencies

Output of **gradle -q dependencies api:dependencies webapp:dependencies**

```
> gradle -q dependencies api:dependencies webapp:dependencies

-----
Root project
-----

No configurations

-----
Project :api - The shared API for the application
-----

compile
\--- org.codehaus.groovy:groovy-all:2.4.7

testCompile
\--- junit:junit:4.12
    \--- org.hamcrest:hamcrest-core:1.3

-----
Project :webapp - The Web application implementation
-----

compile
+--- project :api
|    \--- org.codehaus.groovy:groovy-all:2.4.7
\--- commons-io:commons-io:1.2

testCompile
No dependencies
```

Since a dependency report can get large, it can be useful to restrict the report to a particular configuration. This is achieved with the optional **--configuration** parameter:

Example 4.15. Filtering dependency report by configuration

Output of **gradle -q api:dependencies --configuration testCompile**

```
> gradle -q api:dependencies --configuration testCompile

-----
Project :api - The shared API for the application
-----

testCompile
\--- junit:junit:4.12
    \--- org.hamcrest:hamcrest-core:1.3
```

4.7.5. Listing project buildscript dependencies

Running **gradle buildEnvironment** visualises the buildscript dependencies of the selected project, similarly to how **gradle dependencies** visualises the dependencies of the software being built.

4.7.6. Getting the insight into a particular dependency

Running **gradle dependencyInsight** gives you an insight into a particular dependency (or dependencies) that match specified input. Below is an example of this report:

Example 4.16. Getting the insight into a particular dependency

Output of **gradle -q webapp:dependencyInsight --dependency groovy --configuration**

```
> gradle -q webapp:dependencyInsight --dependency groovy --configuration compile
org.codehaus.groovy:groovy-all:2.4.7
\--- project :api
     \--- compile
```

This task is extremely useful for investigating the dependency resolution, finding out where certain dependencies are coming from and why certain versions are selected. For more information please see the `DependencyInsightReportTask` class in the API documentation.

The built-in `dependencyInsight` task is a part of the 'Help' tasks group. The task needs to be configured with the dependency and the configuration. The report looks for the dependencies that match the specified dependency spec in the specified configuration. If Java related plugins are applied, the `dependencyInsight` task is pre-configured with the 'compile' configuration because typically it's the compile dependencies we are interested in. You should specify the dependency you are interested in via the command line '--dependency' option. If you don't like the defaults you may select the configuration via the '--configuration' option. For more information see the `DependencyInsightReportTask` class in the API documentation.

4.7.7. Listing project properties

Running **gradle properties** gives you a list of the properties of the selected project. This is a snippet from the output:

Example 4.17. Information about properties

Output of **gradle -q api:properties**

```
> gradle -q api:properties

-----
Project :api - The shared API for the application
-----

allprojects: [project ':api']
ant: org.gradle.api.internal.project.DefaultAntBuilder@12345
antBuilderFactory: org.gradle.api.internal.project.DefaultAntBuilderFactory@12345
artifacts: org.gradle.api.internal.artifacts.dsl.DefaultArtifactHandler_Decorated@12
asDynamicObject: DynamicObject for project ':api'
baseClassLoaderScope: org.gradle.api.internal.initialization.DefaultClassLoaderScope
buildDir: /home/user/gradle/samples/userguide/tutorial/projectReports/api/build
buildFile: /home/user/gradle/samples/userguide/tutorial/projectReports/api/build.gra
```

4.7.8. Profiling a build

The **--profile** command line option will record some useful timing information while your build is running and write a report to the `build/reports/profile` directory. The report will be named using the time when the build was run.

This report lists summary times and details for both the configuration phase and task execution. The times for configuration and task execution are sorted with the most expensive operations first. The task execution results also indicate if any tasks were skipped (and the reason) or if tasks that were not skipped did no work.

Builds which utilize a `buildSrc` directory will generate a second profile report for `buildSrc` in the `buildSrc/bui` directory.

Profiled with tasks: -xtest build		
Summary		Task
Total Build Time	2:01.164	:docs
Startup	0.313	:docs:userguideSingleHt
Settings and BuildSrc	4.078	:docs:userguidePdf
Loading Projects	0.074	:docs:checkstyleApi
Configuring Projects	3.208	:docs:userguideStyleShee
Total Task Execution	1:52.671	:docs:groovyd
		:docs:samples
		:docs:javadoc
		:docs:userguideFragment
		:docs:distDocs
		:docs:samplesDocs
		:docs:userguideXhtml
		:docs:userguideHtml
		:docs:userguideDocbook
		:docs:remoteUserguideD
		:docs:samplesDocbook
		:docs:docs
		:docs:userguide
		:core
		:core:compileTestGroovy
		:core:codenarcTest
		:core:checkstyleMain
		:core:compileTestJava

4.8. 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, if you run **“gradle -m clean compile”**, you'll see all the tasks that would be executed as part of the `clean` and `compile` tasks. This is complementary to the `tasks` task, which shows you the tasks which are available for execution.

4.9. 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 D, *Gradle Command Line*.

5

The Gradle Wrapper

Most tools require installation on your computer before you can use them. If the installation is easy, you may think that's fine. But it can be an unnecessary burden on the users of the build. Equally importantly, will the user install the right version of the tool for the build? What if they're building an old version of the software?

The Gradle Wrapper (henceforth referred to as the "Wrapper") solves both these problems and is the preferred way of starting a Gradle build.

5.1. Executing a build with the Wrapper

If a Gradle project has set up the Wrapper (and we recommend all projects do so), you can execute the build using one of the following commands from the root of the project:

- `./gradlew <task>` (on Unix-like platforms such as Linux and Mac OS X)
- `gradlew <task>` (on Windows using the `gradlew.bat` batch file)

Each Wrapper is tied to a specific version of Gradle, so when you first run one of the commands above for a given Gradle version, it will download the corresponding Gradle distribution and use it to execute the build.

Not only does this mean that you don't have to manually install Gradle yourself, but you are also sure to use the version of Gradle that the build is designed for. This makes your historical builds more reliable. Just use the appropriate syntax from above whenever you see a command line starting with **gradle ...** in the user guide, on Stack Overflow, in articles or wherever.

For completeness sake, and to ensure you don't delete any important files, here are the files and directories in a Gradle project that make up the Wrapper:

- `gradlew` (Unix Shell script)
- `gradlew.bat` (Windows batch file)
- `gradle/wrapper/gradle-wrapper.jar` (Wrapper JAR)
- `gradle/wrapper/gradle-wrapper.properties` (Wrapper properties)

If you're wondering where the Gradle distributions are stored, you'll find them in your user home directory under `$USER_HOME/.gradle/wrapper/dists`.

IDEs

When importing a Gradle project via its wrapper, your IDE may ask to use the Gradle 'all' distribution. This is perfectly fine and helps the IDE provide code completion for the build files.

5.2. Adding the Wrapper to a project

The Wrapper is something you *should* check into version control. By distributing the Wrapper with your project, anyone can work with it without needing to install Gradle beforehand. Even better, users of the build are guaranteed to use the version of Gradle that the build was designed to work with. Of course, this is also great for continuous integration servers (i.e. servers that regularly build your project) as it requires no configuration on the server.

You install the Wrapper into your project by running the `wrapper` task. (This task is always available, even if you don't add it to your build). To specify a Gradle version use `--gradle-version` on the command-line. By default, the Wrapper will use a `bin` distribution. This is the smallest Gradle distribution. Some tools, like Android Studio and IntelliJ IDEA, provide additional context information when used with the `all` distribution. You may select a different Gradle distribution type by using `--distribution-type`. You can also set the URL to download Gradle from directly via `--gradle-distribution-url`. If no version or distribution URL is specified, the Wrapper will be configured to use the gradle version the `wrapper` task is executed with. So if you run the `wrapper` task with Gradle 2.4, then the Wrapper configuration will default to version 2.4.

Example 5.1. Running the Wrapper task

Output of **gradle wrapper --gradle-version 2.0**

```
> gradle wrapper --gradle-version 2.0
:wrapper

BUILD SUCCESSFUL

Total time: 1 secs
```

The Wrapper can be further customized by adding and configuring a Wrapper task in your build script, and then executing it.

Example 5.2. Wrapper task

build.gradle

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

After such an execution you find the following new or updated files in your project directory (in case the default configuration of the Wrapper task is used).

Example 5.3. Wrapper generated files

Build layout

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

All of these files *should* be submitted to your version control system. This only needs to be done once. After these files have been added to the project, the project should then be built with the added **gradlew** command. 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 you want to take advantage of new functionality in the Gradle wrapper, then you would need to regenerate the wrapper files.

5.3. Configuration

If you run Gradle with **gradlew**, the Wrapper checks if a Gradle distribution for the Wrapper is available. If so, it delegates to the **gradle** command of this distribution with all the arguments passed originally to the **gradlew** command. If it didn't find a Gradle distribution, it will download it first.

When you configure the `Wrapper` task, you can specify the Gradle version you wish to use. The **gradlew** command will download the appropriate distribution from the Gradle repository. Alternatively, you can specify the download URL of the Gradle distribution. The **gradlew** command will use this URL to download the distribution. If you specified neither a Gradle version nor download URL, the **gradlew** command will download whichever version of Gradle was used to generate the Wrapper files.

For the details on how to configure the Wrapper, see the `Wrapper` class in the API documentation.

If you don't want any download to happen when your project is built via **gradlew**, simply add the Gradle distribution zip to your version control at the location specified by your Wrapper configuration. A relative URL is supported - you can specify a distribution file relative to the location of `gradle-wrapper.properties` file.

If you build via the Wrapper, any existing Gradle distribution installed on the machine is ignored.

5.4. Authenticated Gradle distribution download

The Gradle Wrapper can download Gradle distributions from servers using HTTP Basic Authentication. This enables you to host the Gradle distribution on a private protected server. You can specify a username and password in two different ways depending on your use case: as system properties or directly

Security Warning

HTTP Basic Authentication should only be used with HTTPS

embedded in the `distributionUrl`. Credentials in system properties take precedence over the ones embedded in `distributionUrl`.

URLs and not plain HTTP ones. With Basic Authentication, the user credentials are sent in clear text.

Using system properties can be done in the `.gradle/gradle.properties` file in the user's home directory, or by other means, see Section 12.1, “Configuring the build environment via `gradle.properties`”.

Example 5.4. Specifying the HTTP Basic Authentication credentials using system properties

gradle.properties

```
systemProp.gradle.wrapperUser=username
systemProp.gradle.wrapperPassword=password
```

Embedding credentials in the `distributionUrl` in the `gradle/wrapper/gradle-wrapper.properties` file also works. Please note that this file is to be committed into your source control system. Shared credentials embedded in `distributionUrl` should only be used in a controlled environment.

Example 5.5. Specifying the HTTP Basic Authentication credentials in `distributionUrl`

gradle-wrapper.properties

```
distributionUrl=https://username:password@somehost/path/to/gradle-distribution.zip
```

This can be used in conjunction with a proxy, authenticated or not. See Section 12.3, “Accessing the web via a proxy” for more information on how to configure the Wrapper to use a proxy.

5.5. Verification of downloaded Gradle distributions

The Gradle Wrapper allows for verification of the downloaded Gradle distribution via SHA-256 hash sum comparison. This increases security against targeted attacks by preventing a man-in-the-middle attacker from tampering with the downloaded Gradle distribution.

To enable this feature you'll want to first calculate the SHA-256 hash of a known Gradle distribution. You can generate a SHA-256 hash from Linux and OSX or Windows (via Cygwin) with the **shasum** command.

Example 5.6. Generating a SHA-256 hash

```
> shasum -a 256 gradle-2.4-all.zip
371cb9fbbebbe9880d147f59bab36d61eee122854ef8c9ee1ecf12b82368bcf10 gradle-2.4-all.zip
```

Add the returned hash sum to the `gradle-wrapper.properties` using the `distributionSha256Sum` property.

Example 5.7. Configuring SHA-256 checksum verification

gradle-wrapper.properties

```
distributionSha256Sum=371cb9fbbebbe9880d147f59bab36d61eee122854ef8c9ee1ecf12b8236
```

5.6. Unix file permissions

The Wrapper task adds appropriate file permissions to allow the execution of 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`”.

6

The Gradle Daemon

From Wikipedia...

A daemon is a computer program that runs as a background process, rather than being under the direct control of an interactive user.

Gradle runs on the Java Virtual Machine (JVM) and uses several supporting libraries that require a non-trivial initialization time. As a result, it can sometimes seem a little slow to start. The solution to this problem is the Gradle *Daemon*: a long-lived background process that executes your builds much more quickly than would otherwise be the case. We accomplish this by avoiding the expensive bootstrapping process as well as leveraging caching, by keeping data about your project in memory. Running Gradle builds with the Daemon is no different than without. Simply configure whether you want to use it or not - everything else is handled transparently by Gradle.

6.1. Why the Gradle Daemon is important for performance

The Daemon is a long-lived process, so not only are we able to avoid the cost of JVM startup for every build, but we are able to cache information about project structure, files, tasks, and more in memory.

The reasoning is simple: improve build speed by reusing computations from previous builds. However, the benefits are dramatic: we typically measure build times reduced by 15-75% on subsequent builds. We recommend profiling your build by using `--profile` to get a sense of how much impact the Gradle Daemon can have for you.

The Gradle Daemon is enabled by default starting with Gradle 3.0, so you don't have to do anything to benefit from it.

6.2. Running Daemon Status

To get a list of running Gradle Daemons and their statuses use the `--status` command.

Sample output:

PID	VERSION	STATUS
28411	3.0	IDLE
34247	3.0	BUSY

Currently, a given Gradle version can only connect to daemons of the same version. This means the status output will only show Daemons for the version of Gradle being invoked and not for any other versions. Future versions of Gradle will lift this constraint and will show the running Daemons for all versions of Gradle.

6.3. Disabling the Daemon

The Gradle Daemon is enabled by default, and we recommend always enabling it for developers' machines. There are several ways to disable the Daemon, but the most common one is to add the line

```
org.gradle.daemon=false
```

to the file «USER_HOME»/.gradle/gradle.properties, where «USER_HOME» is your home directory. That's typically one of the following, depending on your platform:

- C:\Users\- /Users/<username> (Mac OS X)
- /home/<username> (Linux)

If that file doesn't exist, just create it using a text editor. You can find details of other ways to disable (and enable) the Daemon in Section 6.5, "FAQ" further down. That section also contains more detailed information on how the Daemon works.

Once you have globally enabled the Daemon in this way, all your builds will take advantage of the speed boost, regardless of the version of Gradle a particular build uses.

6.4. Stopping an existing Daemon

As mentioned, the Daemon is a background process. You needn't worry about a build up of Gradle processes on your machine, though. Every Daemon monitors its memory usage compared to total system memory and will stop itself if idle when available system memory is low. If you want to explicitly stop running Daemon processes for any reason, just use the command **gradle --stop**.

This will terminate all Daemon processes that were started with the same version of Gradle used to execute the command. If you

Continuous integration

At the moment, we recommend that you disable the Daemon for Continuous Integration servers as correctness is usually a priority over speed in CI environments. Using a fresh runtime for each build is more reliable since the runtime is *completely* isolated from any previous builds. Additionally, since the Daemon primarily acts

have the Java Development Kit (JDK) installed, you can easily verify that a Daemon has stopped by running the `jps` command. You'll see any running Daemons listed with the name `GradleDaemon` on a developer's machine.

to reduce build startup times, this isn't as critical in CI as it is

6.5. FAQ

6.5.1. How do I disable the Gradle Daemon?

There are two recommended ways to disable the Daemon persistently for an environment:

- Via environment variables: add the flag `-Dorg.gradle.daemon=false` to the `GRADLE_OPTS` environment variable

Via properties file: add `org.gradle.daemon=false` to the `«GRADLE_USER_HOME»/gradle.properties` file

Note, `«GRADLE_USER_HOME»` defaults to `«USER_HOME»/.gradle`, where `«USER_HOME»` is the home directory of the current user. This location can be configured via the `-g` and `--gradle-user-home` command line switches, as well as by the `GRADLE_USER_HOME` environment variable and `org.gradle.userHome` JVM system property.

Both approaches have the same effect. Which one to use is up to personal preference. Most Gradle users choose the second option and add the entry to the user `gradle.properties` file.

On Windows, this command will disable the Daemon for the current user:

```
(if not exist "%USERPROFILE%\.gradle" mkdir "%USERPROFILE%\.gradle") && (echo. > "%USERPROFILE%\.gradle\gradle.properties")
```

On UNIX-like operating systems, the following Bash shell command will disable the Daemon for the current user:

```
mkdir -p ~/.gradle && echo "org.gradle.daemon=false" >> ~/.gradle/gradle.properties
```

Once the Daemon is disabled for a build environment in this way, a Gradle Daemon will not be started unless explicitly requested using the `--daemon` option.

The `--daemon` and `--no-daemon` command line options enable and disable usage of the Daemon for individual build invocations when using the Gradle command line interface. These command line options have the *highest* precedence when considering the build environment. Typically, it is more convenient to enable the Daemon for an environment (e.g. a user account) so that all builds use the Daemon without requiring to remember to supply the `--daemon` option.

6.5.2. Why is there more than one Daemon process on my machine?

There are several reasons why Gradle will create a new Daemon, instead of using one that is already running. The basic rule is that Gradle will start a new Daemon if there are no existing idle or compatible Daemons available. Gradle will kill any Daemon that has been idle for 3 hours or more, so you don't have to worry about cleaning them up manually.

idle

An idle Daemon is one that is not currently executing a build or doing other useful work.

compatible

A compatible Daemon is one that can (or can be made to) meet the requirements of the requested build environment. The Java runtime used to execute the build is an example aspect of the build environment. Another example is the set of JVM system properties required by the build runtime.

Some aspects of the requested build environment may not be met by an Daemon. If the Daemon is running with a Java 7 runtime, but the requested environment calls for Java 8, then the Daemon is not compatible and another must be started. Moreover, certain properties of a Java runtime cannot be changed once the JVM has started. For example, it is not possible to change the memory allocation (e.g. `-Xmx1024m`), default text encoding, default locale, etc of a running JVM.

The “requested build environment” is typically constructed implicitly from aspects of the build client’s (e.g. Gradle command line client, IDE etc.) environment and explicitly via command line switches and settings. See Chapter 12, *The Build Environment* for details on how to specify and control the build environment.

The following JVM system properties are effectively immutable. If the requested build environment requires any of these properties, with a different value than a Daemon’s JVM has for this property, the Daemon is not compatible.

- `file.encoding`
- `user.language`
- `user.country`
- `user.variant`
- `java.io.tmpdir`
- `javax.net.ssl.keyStore`
- `javax.net.ssl.keyStorePassword`
- `javax.net.ssl.keyStoreType`
- `javax.net.ssl.trustStore`
- `javax.net.ssl.trustStorePassword`
- `javax.net.ssl.trustStoreType`
- `com.sun.management.jmxremote`

The following JVM attributes, controlled by startup arguments, are also effectively immutable. The corresponding attributes of the requested build environment and the Daemon’s environment must match exactly in order for a Daemon to be compatible.

- The maximum heap size (i.e. the `-Xmx` JVM argument)
- The minimum heap size (i.e. the `-Xms` JVM argument)

- The boot classpath (i.e. the `-Xbootclasspath` argument)
- The “assertion” status (i.e. the `-ea` argument)

The required Gradle version is another aspect of the requested build environment. Daemon processes are coupled to a specific Gradle runtime. Working on multiple Gradle projects during a session that use different Gradle versions is a common reason for having more than one running Daemon process.

6.5.3. How much memory does the Daemon use and can I give it more?

If the requested build environment does not specify a maximum heap size, the Daemon will use up to 1GB of heap. It will use your the JVM's default minimum heap size. 1GB is more than enough for most builds. Larger builds with hundreds of subprojects, lots of configuration, and source code may require, or perform better, with more memory.

To increase the amount of memory the Daemon can use, specify the appropriate flags as part of the requested build environment. Please see Chapter 12, *The Build Environment* for details.

6.5.4. How can I stop a Daemon?

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

6.5.5. What can go wrong with Daemon?

Considerable engineering effort has gone into making the Daemon robust, transparent and unobtrusive during day to day development. However, Daemon processes can occasionally be corrupted or exhausted. A Gradle build executes arbitrary code from multiple sources. While Gradle itself is designed for and heavily tested with the Daemon, user build scripts and third party plugins can destabilize the Daemon process through defects such as memory leaks or global state corruption.

It is also possible to destabilize the Daemon (and build environment in general) by running builds that do not release resources correctly. This is a particularly poignant problem when using Microsoft Windows as it is less forgiving of programs that fail to close files after reading or writing.

Gradle actively monitors heap usage and attempts to detect when a leak is starting to exhaust the available heap space in the daemon. When it detects a problem, the Gradle daemon will finish the currently running build and proactively restart the daemon on the next build. This monitoring is enabled by default, but can be disabled by setting the `org.gradle.daemon.performance.enable-monitoring` system property to false.

If it is suspected that the Daemon process has become unstable, it can simply be killed. Recall that the `--no-dae` switch can be specified for a build to prevent use of the Daemon. This can be useful to diagnose whether or not the Daemon is actually the culprit of a problem.

6.6. When should I not use the Gradle Daemon?

It is recommended that the Daemon is used in all developer environments. It is recommended to disable the Daemon for Continuous Integration and build server environments.

The Daemon enables faster builds, which is particularly important when a human is sitting in front of the build. For CI builds, stability and predictability is of utmost importance. Using a fresh runtime (i.e. process) for each build is more reliable as the runtime is completely isolated from previous builds.

6.7. Tools & IDEs

The Gradle Tooling API (see Chapter 14, *Embedding Gradle using the Tooling API*), that is used by IDEs and other tools to integrate with Gradle, always use the Gradle Daemon to execute builds. If you are executing Gradle builds from within your IDE you are using the Gradle Daemon and do not need to enable it for your environment.

However, unless you have explicitly enabled the Gradle Daemon for your environment your builds from the command line will not use the Gradle Daemon.

6.8. How does the Gradle Daemon make builds faster?

The Gradle Daemon is a long lived build process. In between builds it waits idly for the next build. This has the obvious benefit of only requiring Gradle to be loaded into memory once for multiple builds, as opposed to once for each build. This in itself is a significant performance optimization, but that's not where it stops.

A significant part of the story for modern JVM performance is runtime code optimization. For example, HotSpot (the JVM implementation provided by Oracle and used as the basis of OpenJDK) applies optimization to code while it is running. The optimization is progressive and not instantaneous. That is, the code is progressively optimized during execution which means that subsequent builds can be faster purely due to this optimization process. Experiments with HotSpot have shown that it takes somewhere between 5 and 10 builds for optimization to stabilize. The difference in perceived build time between the first build and the 10th for a Daemon can be quite dramatic.

The Daemon also allows more effective in memory caching across builds. For example, the classes needed by the build (e.g. plugins, build scripts) can be held in memory between builds. Similarly, Gradle can maintain in-memory caches of build data such as the hashes of task inputs and outputs, used for incremental building.

6.8.1. Potential future enhancements

Currently, the Daemon makes builds faster by effectively supporting in memory caching and by the JVM optimizer making the code faster. In future Gradle versions, the Daemon will become even smarter and perform work *preemptively*. It could, for example, start downloading dependencies immediately after the build script has been edited under the assumption that the build is about to be run and the newly changed or added dependencies will be required.

There are many other ways in that the Gradle Daemon will enable even faster builds in future Gradle versions.

Dependency Management Basics

This chapter introduces some of the basics of dependency management in Gradle.

7.1. What is dependency management?

Very roughly, dependency management is made up of two pieces. Firstly, Gradle needs to know about the things that your project needs to build or run, in order to find them. We call these incoming files the *dependencies* of the project. Secondly, Gradle needs to build and upload the things that your project produces. We call these outgoing files the *publications* of the project. Let's look at these two pieces in more detail:

Most projects are not completely self-contained. They need files built by other projects in order to be compiled or tested and so on. For example, in order to use Hibernate in my project, I need to include some Hibernate jars in the classpath when I compile my source. To run my tests, I might also need to include some additional jars in the test classpath, such as a particular JDBC driver or the Ehcache jars.

These incoming files form the dependencies of the project. Gradle allows you to tell it what the dependencies of your project are, so that it can take care of finding these dependencies, and making them available in your build. The dependencies might need to be downloaded from a remote Maven or Ivy repository, or located in a local directory, or may need to be built by another project in the same multi-project build. We call this process *dependency resolution*.

Note that this feature provides a major advantage over Ant. With Ant, you only have the ability to specify absolute or relative paths to specific jars to load. With Gradle, you simply declare the “names” of your dependencies, and other layers determine where to get those dependencies from. You can get similar behavior from Ant by adding Apache Ivy, but Gradle does it better.

Often, the dependencies of a project will themselves have dependencies. For example, Hibernate core requires several other libraries to be present on the classpath with it runs. So, when Gradle runs the tests for your project, it also needs to find these dependencies and make them available. We call these *transitive dependencies*.

The main purpose of most projects is to build some files that are to be used outside the project. For example, if your project produces a Java library, you need to build a jar, and maybe a source jar and some documentation, and publish them somewhere.

These outgoing files form the publications of the project. Gradle also takes care of this important work for you. You declare the publications of your project, and Gradle take care of building them and publishing them somewhere. Exactly what “publishing” means depends on what you want to do. You might want to

copy the files to a local directory, or upload them to a remote Maven or Ivy repository. Or you might use the files in another project in the same multi-project build. We call this process *publication*.

7.2. Declaring your dependencies

Let's look at some dependency declarations. Here's a basic build script:

Example 7.1. Declaring dependencies

build.gradle

```
apply plugin: 'java'

repositories {
    mavenCentral()
}

dependencies {
    compile group: 'org.hibernate', name: 'hibernate-core', version: '3.6.7.Final'
    testCompile group: 'junit', name: 'junit', version: '4.+'
}
```

What's going on here? This build script says a few things about the project. Firstly, it states that Hibernate core 3.6.7.Final is required to compile the project's production source. By implication, Hibernate core and its dependencies are also required at runtime. The build script also states that any junit \geq 4.0 is required to compile the project's tests. It also tells Gradle to look in the Maven central repository for any dependencies that are required. The following sections go into the details.

7.3. Dependency configurations

In Gradle dependencies are grouped into *configurations*. A configuration is simply a named set of dependencies. We will refer to them as *dependency configurations*. You can use them to declare the external dependencies of your project. As we will see later, they are also used to declare the publications of your project.

The Java plugin defines a number of standard configurations. These configurations represent the classpaths that the Java plugin uses. Some are listed below, and you can find more details in Table 47.5, “Java plugin - dependency configurations”.

compile

The dependencies required to compile the production source of the project.

runtime

The dependencies required by the production classes at runtime. By default, also includes the compile time dependencies.

testCompile

The dependencies required to compile the test source of the project. By default, also includes the compiled production classes and the compile time dependencies.

testRuntime

The dependencies required to run the tests. By default, also includes the compile, runtime and test compile dependencies.

Various plugins add further standard configurations. You can also define your own custom configurations to use in your build. Please see Section 25.3, “Dependency configurations” for the details of defining and customizing dependency configurations.

7.4. External dependencies

There are various types of dependencies that you can declare. One such type is an *external dependency*. This is a dependency on some files built outside the current build, and stored in a repository of some kind, such as Maven central, or a corporate Maven or Ivy repository, or a directory in the local file system.

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

Example 7.2. Definition of an external dependency

build.gradle

```
dependencies {  
    compile group: 'org.hibernate', name: 'hibernate-core', version: '3.6.7.Final'  
}
```

An external dependency is identified using `group`, `name` and `version` attributes. Depending on which kind of repository you are using, `group` and `version` may be optional.

The shortcut form for declaring external dependencies looks like “`group:name:version`”.

Example 7.3. Shortcut definition of an external dependency

build.gradle

```
dependencies {  
    compile 'org.hibernate:hibernate-core:3.6.7.Final'  
}
```

To find out more about defining and working with dependencies, have a look at Section 25.4, “How to declare your dependencies”.

7.5. Repositories

How does Gradle find the files for external dependencies? Gradle looks for them in a *repository*. A repository is really just a collection of files, organized by `group`, `name` and `version`. Gradle understands several different repository formats, such as Maven and Ivy, and several different ways of accessing the repository, such as using the local file system or HTTP.

By default, Gradle does not define any repositories. You need to define at least one before you can use external dependencies. One option is use the Maven central repository:

Example 7.4. Usage of Maven central repository

build.gradle

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

Or Bintray's JCenter:

Example 7.5. Usage of JCenter repository

build.gradle

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

Or a any other remote Maven repository:

Example 7.6. Usage of a remote Maven repository

build.gradle

```
repositories {  
    maven {  
        url "http://repo.mycompany.com/maven2"  
    }  
}
```

Or a remote Ivy repository:

Example 7.7. Usage of a remote Ivy directory

build.gradle

```
repositories {  
    ivy {  
        url "http://repo.mycompany.com/repo"  
    }  
}
```

You can also have repositories on the local file system. This works for both Maven and Ivy repositories.

Example 7.8. Usage of a local Ivy directory

build.gradle

```
repositories {  
    ivy {  
        // URL can refer to a local directory  
        url "../local-repo"  
    }  
}
```

A project can have multiple repositories. Gradle will look for a dependency in each repository in the order they are specified, stopping at the first repository that contains the requested module.

To find out more about defining and working with repositories, have a look at Section 25.6, “Repositories”.

7.6. Publishing artifacts

Dependency configurations are also used to publish files.^[2] We call these files *publication artifacts*, or usually just *artifacts*.

The plugins do a pretty good job of defining the artifacts of a project, so you usually don't need to do anything special to tell Gradle what needs to be published. However, you do need to tell Gradle where to publish the artifacts. You do this by attaching repositories to the `uploadArchives` task. Here's an example of publishing to a remote Ivy repository:

Example 7.9. Publishing to an Ivy repository

build.gradle

```
uploadArchives {
    repositories {
        ivy {
            credentials {
                username "username"
                password "pw"
            }
            url "http://repo.mycompany.com"
        }
    }
}
```

Now, when you run **gradle uploadArchives**, Gradle will build and upload your Jar. Gradle will also generate and upload an `ivy.xml` as well.

You can also publish to Maven repositories. The syntax is slightly different.^[3] Note that you also need to apply the Maven plugin in order to publish to a Maven repository. when this is in place, Gradle will generate and upload a `pom.xml`.

Example 7.10. Publishing to a Maven repository

build.gradle

```
apply plugin: 'maven'

uploadArchives {
    repositories {
        mavenDeployer {
            repository(url: "file://localhost/tmp/myRepo/")
        }
    }
}
```

To find out more about publication, have a look at Chapter 32, *Publishing artifacts*.

7.7. Where to next?

For all the details of dependency resolution, see Chapter 25, *Dependency Management*, and for artifact publication see Chapter 32, *Publishing artifacts*.

If you are interested in the DSL elements mentioned here, have a look at `Project.configurations{}`, `Project.repositories{}` and `Project.dependencies{}`.

Otherwise, continue on to some of the other tutorials.

[2] We think this is confusing, and we are gradually teasing apart the two concepts in the Gradle DSL.

[3] We are working to make the syntax consistent for resolving from and publishing to Maven repositories.

Introduction to multi-project builds

Only the smallest of projects has a single build file and source tree, unless it happens to be a massive, monolithic application. It's often much easier to digest and understand a project that has been split into smaller, inter-dependent modules. The word "inter-dependent" is important, though, and is why you typically want to link the modules together through a single build.

Gradle supports this scenario through *multi-project* builds.

8.1. Structure of a multi-project build

Such builds come in all shapes and sizes, but they do have some common characteristics:

- A `settings.gradle` file in the root or master directory of the project
- A `build.gradle` file in the root or master directory
- Child directories that have their own `*.gradle` build files (some multi-project builds may omit child project build scripts)

The `settings.gradle` file tells Gradle how the project and subprojects are structured. Fortunately, you don't have to read this file simply to learn what the project structure is as you can run the command `gradle projects`. Here's the output from using that command on the Java *multiproject* build in the Gradle samples:

Example 8.1. Listing the projects in a build

Output of `gradle -q projects`

```
> gradle -q projects
```

```
-----
Root project
-----
```

```
Root project 'multiproject'
+--- Project ':api'
+--- Project ':services'
|    +--- Project ':services:shared'
|    \--- Project ':services:webservice'
\--- Project ':shared'
```

To see a list of the tasks of a project, run `gradle <project-path>:tasks`. For example, try running `gradle :api:tasks`.

This tells you that *multiproject* has three immediate child projects: *api*, *services* and *shared*. The *services*

project then has its own children, *shared* and *webservice*. These map to the directory structure, so it's easy to find them. For example, you can find *webservice* in `<root>/services/webservice`.

By default, Gradle uses the name of the directory it finds the `settings.gradle` as the name of the root project. This usually doesn't cause problems since all developers check out the same directory name when working on a project. On Continuous Integration servers, like Jenkins, the directory name may be auto-generated and not match the name in your VCS. For that reason, it's recommended that you always set the root project name to something predictable, even in single project builds. You can configure the root project name by setting `rootProject.name`.

Each project will usually have its own build file, but that's not necessarily the case. In the above example, the *services* project is just a container or grouping of other subprojects. There is no build file in the corresponding directory. However, *multiproject* does have one for the root project.

The root `build.gradle` is often used to share common configuration between the child projects, for example by applying the same sets of plugins and dependencies to all the child projects. It can also be used to configure individual subprojects when it is preferable to have all the configuration in one place. This means you should always check the root build file when discovering how a particular subproject is being configured.

Another thing to bear in mind is that the build files might not be called `build.gradle`. Many projects will name the build files after the subproject names, such as `api.gradle` and `services.gradle` from the previous example. Such an approach helps a lot in IDEs because it's tough to work out which `build.gradle` file out of twenty possibilities is the one you want to open. This little piece of magic is handled by the `settings` file, but as a build user you don't need to know the details of how it's done. Just have a look through the child project directories to find the files with the `.gradle` suffix.

Once you know what subprojects are available, the key question for a build user is how to execute the tasks within the project.

8.2. Executing a multi-project build

From a user's perspective, multi-project builds are still collections of tasks you can run. The difference is that you may want to control *which* project's tasks get executed. You have two options here:

- Change to the directory corresponding to the subproject you're interested in and just execute **gradle <task>** as normal.
- Use a qualified task name from any directory, although this is usually done from the root. For example: **gradle <task>** will build the *webservice* subproject and any subprojects it depends on.

The first approach is similar to the single-project use case, but Gradle works slightly differently in the case of a multi-project build. The command **gradle test** will execute the `test` task in any subprojects, relative to the current working directory, that have that task. So if you run the command from the root project directory, you'll run `test` in *api*, *shared*, *services:shared* and *services:webservice*. If you run the command from the *services* project directory, you'll only execute the task in *services:shared* and *services:webservice*.

For more control over what gets executed, use qualified names (the second approach mentioned). These are

paths just like directory paths, but use ‘:’ instead of ‘/’ or ‘\’. If the path begins with a ‘:’, then the path is resolved relative to the root project. In other words, the leading ‘:’ represents the root project itself. All other colons are path separators.

This approach works for any task, so if you want to know what tasks are in a particular subproject, just use the `tasks` task, e.g. **gradle :services:webservice:tasks**.

Regardless of which technique you use to execute tasks, Gradle will take care of building any subprojects that the target depends on. You don’t have to worry about the inter-project dependencies yourself. If you’re interested in how this is configured, you can read about writing multi-project builds later in the user guide.

There’s one last thing to note. When you’re using the Gradle wrapper, the first approach doesn’t work well because you have to specify the path to the wrapper script if you’re not in the project root. For example, if you’re in the webservice subproject directory, you would have to run **../../gradlew build**.

That’s all you really need to know about multi-project builds as a build user. You can now identify whether a build is a multi-project one and you can discover its structure. And finally, you can execute tasks within specific subprojects.

Continuous build

Continuous build is an incubating feature. This means that it is incomplete and not yet at regular Gradle production quality. This also means that this Gradle User Guide chapter is a work in progress.

Typically, you ask Gradle to perform a single build by way of specifying tasks that Gradle should execute. Gradle will determine the the actual set of tasks that need to be executed to satisfy the request, execute them all, and then stop doing work until the next request. A continuous build differs in that Gradle will keep satisfying the initial build request (until instructed to stop) by executing the build when it is detected that the result of the previous build is now out of date. For example, if your build compiles Java source files to Java class files, a continuous build would automatically initiate a compile when the source files change. Continuous build is useful for many scenarios.

9.1. How do I start and stop a continuous build?

A continuous build can be started by supplying either the `--continuous` or `-t` switches to Gradle, along with the list of tasks, switches and arguments that define the work to do. For example, `gradle build --cont`. This will have the same effect as running `gradle build`, but instead of Gradle exiting when done, it will wait for changes to the build inputs. When a change occurs, `gradle build` will be automatically executed again and the process repeats.

If Gradle is attached to an interactive input source, such as a terminal, the continuous build can be exited by pressing **CTRL-D** (On Microsoft Windows, it is required to also press **ENTER** or **RETURN** after **CTRL-D**). If Gradle is not attached to an interactive input source (e.g. is running as part of a script), the build process must be terminated (e.g. using the `kill` command or similar). If the build is being executed via the Tooling API, the build can be cancelled using the Tooling API's cancellation mechanism.

9.2. What will cause a subsequent build?

At this time, only changes to task inputs are noticed. Gradle will start watching for changes just before the task starts to execute. No other changes will initiate a build. For example, changes to build scripts and build logic will not initiate build. Likewise, changes to files that are read during the configuration of the

Task file inputs

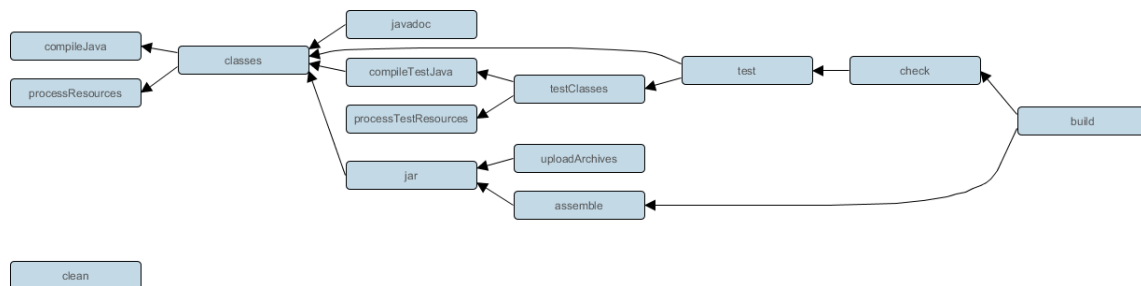
Task implementations declare their file system inputs by annotating their properties with `InputFiles` and other similar

build, not the execution, will not initiate a build. In order to incorporate such changes, the continuous build must be restarted manually.

annotations. Please see Section 19.9, “Up-to-date checks (AKA Incremental Build)” for more information.

Consider a typical build using the Java plugin, using the conventional filesystem layout. The following diagram visualizes the task graph for **gradle build**:

Figure 9.1. Java plugin task graph



The following key tasks of the graph use files in the corresponding directories as inputs:

compileJava

`src/main/java`

processResources

`src/main/resources`

compileTestJava

`src/test/java`

processTestResources

`src/test/resources`

Assuming that the initial build is successful (i.e. the `build` task and its dependencies complete without error), changes to files in, or the addition/remove of files from, the locations listed above will initiate a new build. If a change is made to a Java source file in `src/main/java`, the build will fire and all tasks will be scheduled. Gradle's incremental build support ensures that only the tasks that are actually affected by the change are executed.

If the change to the main Java source causes compilation to fail, subsequent changes to the test source in `src/test` will not initiate a new build. As the test source depends on the main source, there is no point building until the main source has changed, potentially fixing the compilation error. After each build, only the inputs of the tasks that actually executed will be monitored for changes.

Continuous build is in no way coupled to compilation. It works for all types of tasks. For example, the `processResources` task copies and processes the files from `src/main/resources` for inclusion in the built JAR. As such, a change to any file in this directory will also initiate a build.

9.3. Limitations and quirks

There are several issues to be aware with the current implementation of continuous build. These are likely to be addressed in future Gradle releases.

9.3.1. Build cycles

Gradle starts watching for changes just before a task executes. If a task modifies its own inputs while executing, Gradle will detect the change and trigger a new build. If every time the task executes, the inputs are modified again, the build will be triggered again. This isn't unique to continuous build. A task that modifies its own inputs will never be considered up-to-date when run "normally" without continuous build.

If your build enters a build cycle like this, you can track down the task by looking at the list of files reported changed by Gradle. After identifying the file(s) that are changed during each build, you should look for a task that has that file as an input. In some cases, it may be obvious (e.g., a Java file is compiled with `compileJava`). In other cases, you can use `--info` logging to find the task that is out-of-date due to the identified files.

9.3.2. Restrictions with Java 9

Due to class access restrictions related to Java 9, Gradle cannot set some operating system specific options, which means that:

- On Mac OS X, Gradle will poll for file changes every 10 seconds instead of every 2 seconds.
- On Windows, Gradle must use individual file watches (like on Linux/Mac OS), which may cause continuous build to no longer work on very large projects.

9.3.3. Performance and stability

The JDK file watching facility relies on inefficient file system polling on Mac OS X (see: [JDK-7133447](#)). This can significantly delay notification of changes on large projects with many source files.

Additionally, the watching mechanism may deadlock under *heavy* load on Mac OS X (see: [JDK-8079620](#)). This will manifest as Gradle appearing not to notice file changes. If you suspect this is occurring, exit continuous build and start again.

On Linux, OpenJDK's implementation of the file watch service can sometimes miss file system events (see: [JDK-8145981](#)).

9.3.4. Changes to symbolic links

- Creating or removing symbolic link to files will initiate a build.
- Modifying the target of a symbolic link will not cause a rebuild.
- Creating or removing symbolic links to directories will not cause rebuilds.
- Creating new files in the target directory of a symbolic link will not cause a rebuild.
- Deleting the target directory will not cause a rebuild.

9.3.5. Changes to build logic are not considered

The current implementation does not recalculate the build model on subsequent builds. This means that changes to task configuration, or any other change to the build model, are effectively ignored.

10

Composite builds

Composite build is an incubating feature. While useful for many use cases, there are bugs to be discovered, rough edges to smooth, and enhancements we plan to make. Thanks for trying it out!

10.1. What is a composite build?

A composite build is simply a build that includes other builds. In many ways a composite build is similar to a Gradle multi-project build, except that instead of including single projects, complete builds are included.

Composite builds allow you to:

- combine builds that are usually developed independently, for instance when trying out a bug fix in a library that your application uses
- decompose a large multi-project build into smaller, more isolated chunks that can be worked in independently or together as needed

A build that is included in a composite build is referred to, naturally enough, as an "included build". Included builds do not share any configuration with the composite build, or the other included builds. Each included build is configured and executed in isolation.

Included builds interact with other builds via `dependency substitution`. If any build in the composite has a dependency that can be satisfied by the included build, then that dependency will be replaced by a project dependency on the included build.

By default, Gradle will attempt to determine the dependencies that can be substituted by an included build. However for more flexibility, it is possible to explicitly declare these substitutions if the default ones determined by Gradle are not correct for the composite. See Section 10.3, "Declaring the dependencies substituted by an included build".

As well as consuming outputs via project dependencies, a composite build can directly declare task dependencies on included builds. Included builds are isolated, and are not able to declare task dependencies on the composite build or on other included builds. See Section 10.4, "Depending on tasks in an included build".

10.2. Defining a composite build

The following examples demonstrate the various ways that 2 Gradle builds that are normally developed separately can be combined into a composite build. For these examples, the `my-utils` multi-project build produces 2 different java libraries (`number-utils` and `string-utils`), and the `my-app` build produces an executable using functions from those libraries.

The `my-app` build does not have direct dependencies on `my-utils`. Instead, it declares binary dependencies on the libraries produced by `my-utils`.

Example 10.1. Dependencies of my-app

`my-app/build.gradle`

```
apply plugin: 'java'
apply plugin: 'application'
apply plugin: 'idea'

group "org.sample"
version "1.0"

mainClassName = "org.sample.myapp.Main"

dependencies {
    compile "org.sample:number-utils:1.0"
    compile "org.sample:string-utils:1.0"
}

repositories {
    jcenter()
}
```

Note: The code for this example can be found at `samples/compositeBuilds/basic` in the ‘-all’ distribution of Gradle.

10.2.1. Defining a composite build via `--include-build`

The `--include-build` command-line argument turns the executed build into a composite, substituting dependencies from the included build into the executed build.

Example 10.2. Declaring a command-line composite

Output of **gradle --include-build ../my-utils run**

```
> gradle --include-build ../my-utils run
[composite-build] Configuring build: /home/user/gradle/samples/compositeBuilds/basic
:compileJava
:my-utils:number-utils:compileJava
:my-utils:number-utils:processResources NO-SOURCE
:my-utils:number-utils:classes
:my-utils:number-utils:jar
:my-utils:string-utils:compileJava
:my-utils:string-utils:processResources NO-SOURCE
:my-utils:string-utils:classes
:my-utils:string-utils:jar
:processResources NO-SOURCE
:classes
:run
The answer is 42

BUILD SUCCESSFUL
```

10.2.2. Defining a composite build via `settings.gradle`

It's possible to make the above arrangement persistent, by using `Settings.includeBuild(java.lang.Object)` to declare the included build in the `settings.gradle` file. The `settings.gradle` file can be used to add subprojects and included builds at the same time. Included builds are added by location. See the examples below for more details.

10.2.3. Defining a separate composite build

One downside of the above approach is that it requires you to modify an existing build, rendering it less useful as a standalone build. One way to avoid this is to define a separate composite build, whose only purpose is to combine otherwise separate builds.

Example 10.3. Declaring a separate composite

settings.gradle

```
rootProject.name='adhoc'

includeBuild '../my-app'
includeBuild '../my-utils'
```

In this scenario, the 'main' build that is executed is the composite, and it doesn't define any useful tasks to execute itself. In order to execute the 'run' task in the 'my-app' build, the composite build must define a delegating task.

Example 10.4. Depending on task from included build

build.gradle

```
task run {
    dependsOn gradle.includedBuild('my-app').task(':run')
}
```

More details tasks that depend on included build tasks below.

10.2.4. Restrictions on included builds

Most builds can be included into a composite, however there are some limitations.

Every included build:

- must have a `settings.gradle` file.
- must not itself be a composite build.
- must not have a `rootProject.name` the same as another included build.
- must not have a `rootProject.name` the same as a top-level project of the composite build.
- must not have a `rootProject.name` the same as the composite build `rootProject.name`.

10.3. Declaring the dependencies substituted by an included build

By default, Gradle will configure each included build in order to determine the dependencies it can provide. The algorithm for doing this is very simple: Gradle will inspect the group and name for the projects in the included build, and substitute project dependencies for any external dependency matching `${project.group}`.

There are cases when the default substitutions determined by Gradle are not sufficient, or they are not correct for a particular composite. For these cases it is possible to explicitly declare the substitutions for an included build. Take for example a single-project build 'unpublished', that produces a java utility library but does not declare a value for the group attribute:

Example 10.5. Build that does not declare group attribute

build.gradle

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

When this build is included in a composite, it will attempt to substitute for the dependency module "undefined:unpublished" ("undefined" being the default value for `project.group`, and 'unpublished' being the root project name). Clearly this isn't going to be very useful in a composite build. To use the unpublished library unmodified in a composite build, the composing build can explicitly declare the substitutions that it provides:

Example 10.6. Declaring the substitutions for an included build

settings.gradle

```
rootProject.name = 'app'

includeBuild('../anonymous-library') {
    dependencySubstitution {
        substitute module('org.sample:number-utils') with project(':')
    }
}
```

With this configuration, the "my-app" composite build will substitute any dependency on `org.sample:number` with a dependency on the root project of "unpublished".

10.3.1. Cases where included build substitutions must be declared

Many builds that use the `uploadArchives` task to publish artifacts will function automatically as an included build, without declared substitutions. Here are some common cases where declared substitutions are required:

- When the `archivesBaseName` property is used to set the name of the published artifact.
- When a configuration other than `default` is published: this usually means a task other than `uploadArchives` is used.
- When the `MavenPom.addFilter()` is used to publish artifacts that don't match the project name.
- When the `maven-publish` or `ivy-publish` plugins are used for publishing, and the publication coordinates don't match `${project.group}:${project.name}`.

10.3.2. Cases where composite build substitutions won't work

Some builds won't function correctly when included in a composite, even when dependency substitutions are explicitly declared. This limitation is due to the fact that a project dependency that is substituted will always point to the `default` configuration of the target project. Any time that the artifacts and dependencies specified for the default configuration of a project don't match what is actually published to a repository, then the composite build may exhibit different behaviour.

Here are some cases where the publish module metadata may be different from the project default configuration:

- When a configuration other than `default` is published.
- When the `maven-publish` or `ivy-publish` plugins are used.
- When the `POM` or `ivy.xml` file is tweaked as part of publication.

Builds using these features function incorrectly when included in a composite build. We plan to improve this in the future.

10.4. Depending on tasks in an included build

While included builds are isolated from one another and cannot declare direct dependencies, a composite build is able to declare task dependencies on its included builds. The included builds are accessed using `Gradle.getIncludedBuilds()` or `Gradle.includedBuild(java.lang.String)`, and a task reference is obtained via the `IncludedBuild.task(java.lang.String)` method.

Using these APIs, it is possible to declare a dependency on a task in a particular included build, or tasks with a certain path in all or some of the included builds.

Example 10.7. Depending on a single task from an included build

build.gradle

```
task run {
    dependsOn gradle.includedBuild('my-app').task(':run')
}
```

Example 10.8. Depending on tasks with path in all included builds

build.gradle

```
task publishDeps {
    dependsOn gradle.includedBuilds*.task(':uploadArchives')
}
```

10.5. Current limitations and future plans for composite builds

We think composite builds are pretty useful already. However, there are some things that don't yet work the way we'd like, and other improvements that we think will make things work even better.

Limitations of the current implementation include:

- No support for included builds that have publications that don't mirror the project default configuration. See Section 10.3.2, “Cases where composite build substitutions won't work”.
- Native builds are not supported. (Binary dependencies are not yet supported for native builds).
- Substituting plugins only works with the `buildscript` block but not with the `plugins` block.

Improvements we have planned for upcoming releases include:

- Better detection of dependency substitution, for build that publish with custom coordinates, builds that produce multiple components, etc. This will reduce the cases where dependency substitution needs to be explicitly declared for an included build.
- The ability to target a task or tasks in an included build directly from the command line. We are currently exploring syntax options for allowing this functionality, which will remove many cases where a delegating task is required in the composite.
- Execution of included builds in parallel.

- Detection of changes to included builds when running with continuous build (`-t`).
- Making the implicit `buildSrc` project an included build.
- Supporting composite-of-composite builds.

11

Using the Gradle Graphical User Interface

The Gradle GUI has been deprecated and will be removed in Gradle 4.0. Consider using an IDE with support for Gradle e.g. Eclipse, IntelliJ or NetBeans instead.

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.

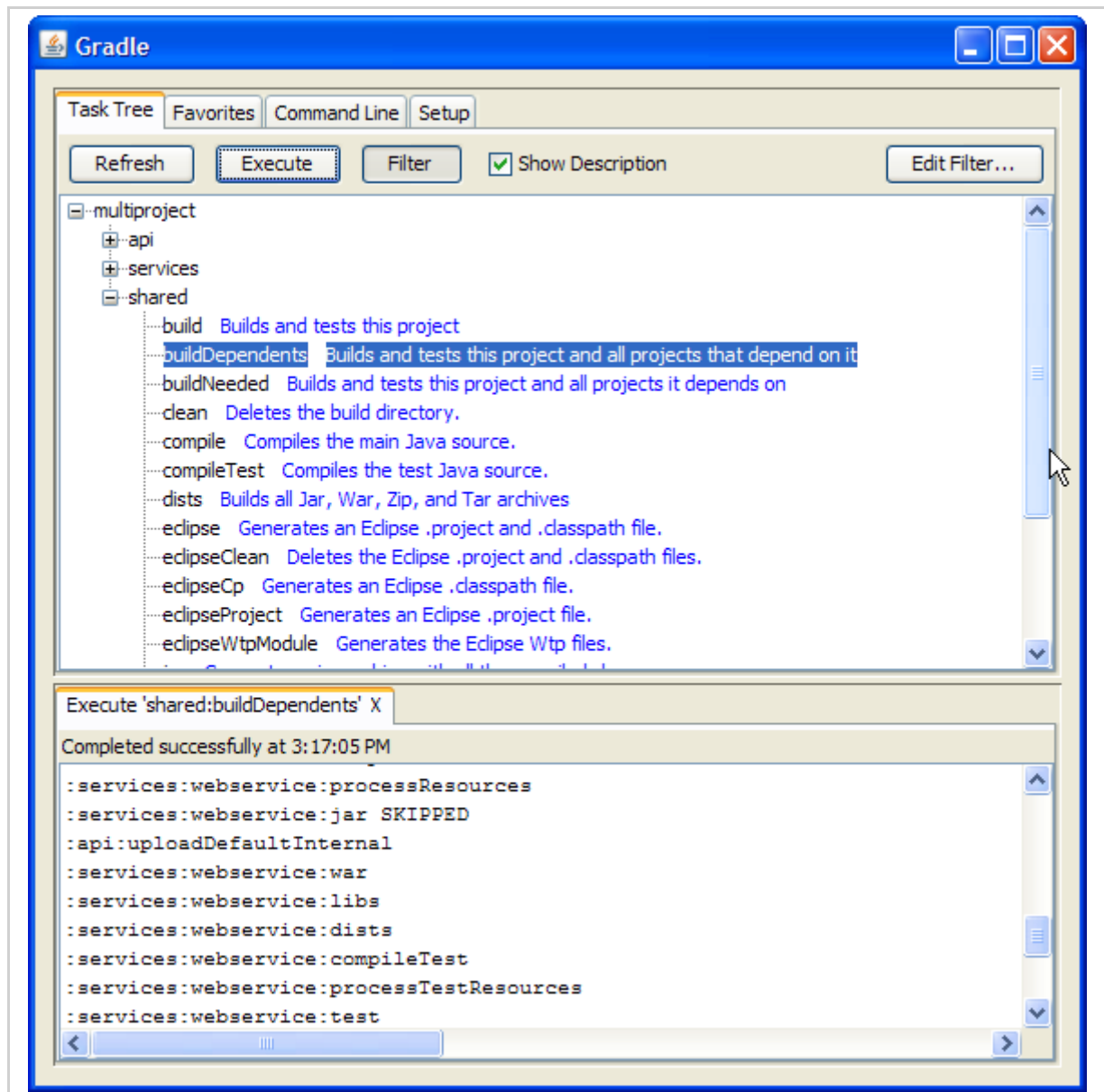
Example 11.1. Launching the GUI

```
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 11.1. 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.

11.1. 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.

11.2. Favorites

The Favorites tab is a good 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 be 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.

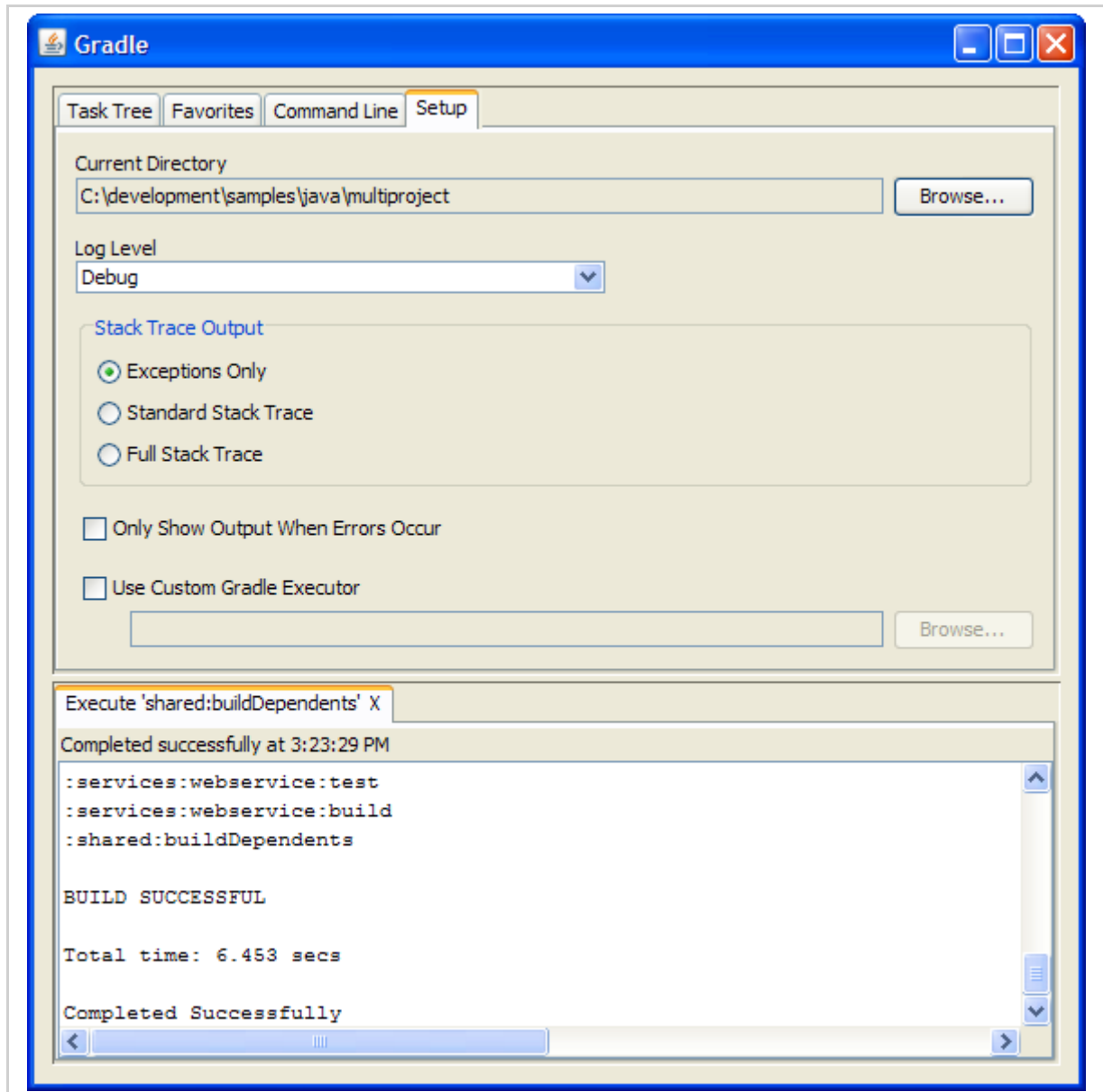
11.3. Command Line

The Command Line tab is where you can 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.

11.4. Setup

The Setup tab allows configuration of some general settings.

Figure 11.2. 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 in 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).

The Build Environment

12.1. Configuring the build environment via `gradle.properties`

Gradle provides several options that make it easy to configure the Java process that will be used to execute your build. While it's possible to configure these in your local environment via `GRADLE_OPTS` or `JAVA_OPTS`, certain settings like JVM memory settings, Java home, daemon on/off can be more useful if they can be versioned with the project in your VCS so that the entire team can work with a consistent environment. Setting up a consistent environment for your build is as simple as placing these settings into a `gradle.properties` file. The configuration is applied in following order (if an option is configured in multiple locations the last one wins):

- from `gradle.properties` in project build dir.
- from `gradle.properties` in gradle user home.
- from system properties, e.g. when `-Dsome.property` is set on the command line.

When setting these properties you should keep in mind that Gradle requires a Java JDK or JRE of version 7 or higher to run.

The following properties can be used to configure the Gradle build environment:

`org.gradle.daemon`

When set to `true` the Gradle daemon is used to run the build. For local developer builds this is our favorite property. The developer environment is optimized for speed and feedback so we nearly always run Gradle jobs with the daemon. We don't run CI builds with the daemon (i.e. a long running process) as the CI environment is optimized for consistency and reliability.

`org.gradle.java.home`

Specifies the Java home for the Gradle build process. The value can be set to either a `jdk` or `jre` location, however, depending on what your build does, `jdk` is safer. A reasonable default is used if the setting is unspecified.

`org.gradle.jvmargs`

Specifies the `jvmargs` used for the daemon process. The setting is particularly useful for tweaking memory settings. At the moment the default settings are pretty generous with regards to memory.

`org.gradle.configureondemand`

Enables new incubating mode that makes Gradle selective when configuring projects. Only relevant projects are configured which results in faster builds for large multi-projects. See the section called

“Configuration on demand”.

`org.gradle.parallel`

When configured, Gradle will run in incubating parallel mode.

`org.gradle.workers.max`

When configured, Gradle will use a maximum of the given number of workers. See `--max-workers` for details.

`org.gradle.debug`

When set to true, Gradle will run the build with remote debugging enabled, listening on port 5005. Note that this is the equivalent of adding `-agentlib:jdwp=transport=dt_socket,server=y,suspend=true` to the JVM command line and will suspend the virtual machine until a debugger is attached.

`org.gradle.daemon.performance.enable-monitoring`

When set to false, Gradle will not monitor the memory usage of running daemons. See Section 6.5.5, “What can go wrong with Daemon?”.

12.1.1. Forked Java processes

Many settings (like the Java version and maximum heap size) can only be specified when launching a new JVM for the build process. This means that Gradle must launch a separate JVM process to execute the build after parsing the various `gradle.properties` files. When running with the daemon, a JVM with the correct parameters is started once and reused for each daemon build execution. When Gradle is executed without the daemon, then a new JVM must be launched for every build execution, unless the JVM launched by the Gradle start script happens to have the same parameters.

This launching of an extra JVM on every build execution is quite expensive, which is why if you are setting either `org.gradle.java.home` or `org.gradle.jvmargs` we highly recommend that you use the Gradle Daemon. See Chapter 6, *The Gradle Daemon* for more details.

12.2. 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 add properties to your project objects using properties files. You can place a `gradle.properties` file in the Gradle user home directory (defined by the “`GRADLE_USER_HOME`” environment variable, which if not set 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 set in a `gradle.properties` file 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.

Gradle can also set project properties when it sees specially-named system properties or environment variables. This feature is very useful when you don't have admin rights to a continuous integration server

and you need to set property values that should not be easily visible, typically for security reasons. In that situation, you can't use the `-P` option, and you can't change the system-level configuration files. The correct strategy is to change the configuration of your continuous integration build job, adding an environment variable setting that matches an expected pattern. This won't be visible to normal users on the system. ^[4]

If the environment variable name looks like `ORG_GRADLE_PROJECT_prop=somevalue`, then Gradle will set a `prop` property on your project object, with the value of `somevalue`. Gradle also supports this for system properties, but with a different naming pattern, which looks like `org.gradle.project.prop`.

You can also set system properties in the `gradle.properties` file. If a property name in such a file has the prefix `“systemProp.”`, like `“systemProp.propName”`, then the property and its value will be set as a system property, without the prefix. In a multi project build, `“systemProp.”` properties set in any project except the root will be ignored. That is, only the root project's `gradle.properties` file will be checked for properties that begin with the `“systemProp.”` prefix.

Example 12.1. Setting properties with a `gradle.properties` file

`gradle.properties`

```
gradlePropertiesProp=gradlePropertiesValue
sysProp=shouldBeOverWrittenBySysProp
envProjectProp=shouldBeOverWrittenByEnvProp
systemProp.system=systemValue
```

`build.gradle`

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

Output of `gradle -q -PcommandLineProjectProp=commandLineProjectPropValue -Dorg.c`

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

12.2.1. 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. If this property does not exist, an exception will be thrown and the build will fail. If your build script relies on optional properties the user might set, perhaps in a `gradle.properties` file, you need to check for existence before you access them. You can do this by using the method `hasProperty('propertyName')` which returns `true` or `false`.

12.3. Accessing the web via a proxy

Configuring an HTTP or HTTPS proxy (for downloading dependencies, for example) is done via standard JVM system properties. These properties can be set directly in the build script; for example, setting the HTTP proxy host would be done with `System.setProperty('http.proxyHost', 'www.somehost.org')`. Alternatively, the properties can be specified in a `gradle.properties` file, either in the build's root directory or in the Gradle home directory.

Example 12.2. Configuring an HTTP 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
```

There are separate settings for HTTPS.

Example 12.3. Configuring an HTTPS proxy

gradle.properties

```
systemProp.https.proxyHost=www.somehost.org
systemProp.https.proxyPort=8080
systemProp.https.proxyUser=userid
systemProp.https.proxyPassword=password
systemProp.https.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's a link to the repository. The other is a Networking Properties page from the JDK docs. If anyone knows of a better overview, please let us know via the mailing list.

12.3.1. NTLM Authentication

If your proxy requires NTLM authentication, you may need to provide the authentication domain as well as the username and password. There are 2 ways that you can provide the domain for authenticating to a NTLM proxy:

- Set the `http.proxyUser` system property to a value like `domain/username`.
- Provide the authentication domain via the `http.auth.ntlm.domain` system property.

[4] *Jenkins*, *Teamcity*, or *Bamboo* are some CI servers which offer this functionality.

13

Troubleshooting

This chapter is currently a work in progress.

When using Gradle (or any software package), you can run into problems. You may not understand how to use a particular feature, or you may encounter a defect. Or, you may have a general question about Gradle.

This chapter gives some advice for troubleshooting problems and explains how to get help with your problems.

13.1. Working through problems

If you are encountering problems, one of the first things to try is using the very latest release of Gradle. New versions of Gradle are released frequently with bug fixes and new features. The problem you are having may have been fixed in a new release.

If you are using the Gradle Daemon, try temporarily disabling the daemon (you can pass the command line switch `--no-daemon`). More information about troubleshooting the daemon process is located in Chapter 6, *The Gradle Daemon*.

13.2. Getting help

The place to go for help with Gradle is <http://forums.gradle.org>. The Gradle Forums is the place where you can report problems and ask questions of the Gradle developers and other community members.

If something's not working for you, posting a question or problem report to the forums is the fastest way to get help. It's also the place to post improvement suggestions or new ideas. The development team frequently posts news items and announces releases via the forum, making it a great way to stay up to date with the latest Gradle developments.

Embedding Gradle using the Tooling API

14.1. Introduction to the Tooling API

Gradle provides a programmatic API called the Tooling API, which you can use for embedding Gradle into your own software. This API allows you to execute and monitor builds and to query Gradle about the details of a build. The main audience for this API is IDE, CI server, other UI authors; however, the API is open for anyone who needs to embed Gradle in their application.

- Gradle TestKit uses the Tooling API for functional testing of your Gradle plugins.
- Eclipse Buildship uses the Tooling API for importing your Gradle project and running tasks.
- IntelliJ IDEA uses the Tooling API for importing your Gradle project and running tasks.

14.2. Tooling API Features

A fundamental characteristic of the Tooling API is that it operates in a version independent way. This means that you can use the same API to work with builds that use different versions of Gradle, including versions that are newer or older than the version of the Tooling API that you are using. The Tooling API is Gradle wrapper aware and, by default, uses the same Gradle version as that used by the wrapper-powered build.

Some features that the Tooling API provides:

- Query the details of a build, including the project hierarchy and the project dependencies, external dependencies (including source and Javadoc jars), source directories and tasks of each project.
- Execute a build and listen to stdout and stderr logging and progress messages (e.g. the messages shown in the 'status bar' when you run on the command line).
- Execute a specific test class or test method.
- Receive interesting events as a build executes, such as project configuration, task execution or test execution.
- Cancel a build that is running.
- Combine multiple separate Gradle builds into a single composite build.
- The Tooling API can download and install the appropriate Gradle version, similar to the wrapper.
- The implementation is lightweight, with only a small number of dependencies. It is also a well-behaved library, and makes no assumptions about your classloader structure or logging configuration. This makes the API easy to embed in your application.

14.3. Tooling API and the Gradle Build Daemon

The Tooling API always uses the Gradle daemon. This means that subsequent calls to the Tooling API, be it model building requests or task executing requests will be executed in the same long-living process. Chapter 6, *The Gradle Daemon* contains more details about the daemon, specifically information on situations when new daemons are forked.

14.4. Quickstart

As the Tooling API is an interface for developers, the Javadoc is the main documentation for it. We provide several *samples* that live in `samples/toolingApi` in your Gradle distribution. These samples specify all of the required dependencies for the Tooling API with examples for querying information from Gradle builds and executing tasks from the Tooling API.

To use the Tooling API, add the following repository and dependency declarations to your build script:

Example 14.1. Using the tooling API

build.gradle

```
repositories {
    maven { url 'https://repo.gradle.org/gradle/libs-releases' }
}

dependencies {
    compile "org.gradle:gradle-tooling-api:${toolingApiVersion}"
    // The tooling API need an SLF4J implementation available at runtime, replace th
    runtime 'org.slf4j:slf4j-simple:1.7.10'
}
```

The main entry point to the Tooling API is the `GradleConnector`. You can navigate from there to find code samples and explore the available Tooling API models. You can use `GradleConnector.connect()` to create a `ProjectConnection`. A `ProjectConnection` connects to a single Gradle project. Using the connection you can execute tasks, tests and retrieve models relative to this project.

14.5. Gradle version and Java version compatibility

The current version of the Tooling API supports running builds using Gradle versions 1.2 and later.

You should note that not all features of the Tooling API are available for all versions of Gradle. For example, build cancellation is only available when a build uses Gradle 2.1 and later. Refer to the documentation for each class and method for more details.

The current Gradle version can be used from Tooling API versions 2.0 or later.

The Tooling API requires Java 7 or later. The Gradle version used by builds may have additional Java version requirements.

15

Build Cache

This feature is a work in progress. There is no public API for enabling or using a build cache; although, you may see references to it while we build the underlying infrastructure.

15.1. Overview

Build caching is a new kind of cache mechanism that aims to save time by reusing outputs produced by other builds.

15.2. Task Output Caching

Beyond incremental builds described in Section 19.9, “Up-to-date checks (AKA Incremental Build)”, Gradle can save time by reusing outputs from previous executions of a task by matching inputs to the task. Task outputs can be reused between builds on one computer or even between builds running on different computers via a build cache.

This feature is a work in progress. There is no public API or documentation for enabling it yet.

15.2.1. Making builds faster

Since a task describes all of its inputs and outputs, Gradle can compute a cache key that uniquely defines the task's outputs based on its inputs. That cache key is used to request previous outputs from a build cache or push new outputs to the build cache. If the previous build is already populated by someone else, e.g. your continuous integration server or other developers, you can avoid executing most tasks locally.

The following inputs contribute to the cache key for a task:

- The task type and its classpath
- The names of the output properties
- The names and values of properties annotated as described in the section called “Custom task types”
- The names and values of properties added by the DSL via `TaskInputs`
- The classpath of the Gradle distribution, `buildSrc` and plugins
- The content of the build script when it affects execution of the task

Task types need to opt-in to task output caching using the `@CacheableTask` annotation. Many built-in Gradle task types are cacheable, but custom task types are not cacheable by default.

Part III. Writing Gradle build scripts

16

Build Script Basics

16.1. Projects and tasks

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

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

Each project is made up of one or more *tasks*. A task represents some atomic piece of work which a build performs. This might be compiling some classes, creating a JAR, generating Javadoc, or publishing some archives to a repository.

For now, we will look at defining some simple tasks in a build with one project. Later chapters will look at working with multiple projects and more about working with projects and tasks.

16.2. Hello world

You run a Gradle build using the **gradle** command. The **gradle** command 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. The build script defines a project and its tasks.

To try this out, create the following build script named `build.gradle`.

Example 16.1. Your first build script

build.gradle

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

In a command-line shell, move to the containing directory and execute the build script with **gradle -q hello** :

Example 16.2. Execution of a build script

Output of **gradle -q hello**

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

What's going on here? This build script 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, you would be right. Gradle tasks are the equivalent to Ant targets, but as you will see, they are much more powerful. We have used a different 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*.

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. This keeps the example output in this user guide a little clearer. You don't need to use this option if you don't want to. See Chapter 24, *Logging* for more details about the command-line options which affect Gradle's output.

16.3. A shortcut task definition

This functionality is deprecated and will be removed in Gradle 5.0 without replacement. Use the methods `Task.doFirst(org.gradle.api.Action)` and `Task.doLast(org.gradle.api.Action)` to define an action instead, as demonstrated by the rest of the examples in this chapter.

There is a shorthand way to define a task like our `hello` task above, which is more concise.

Example 16.3. A task definition shortcut

build.gradle

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

Again, this defines a task called `hello` with a single closure to execute. The `<<` operator is simply an alias for `doLast`.

16.4. Build scripts are code

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

Example 16.4. Using Groovy in Gradle's tasks

build.gradle

```
task upper {
    doLast {
        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 16.5. Using Groovy in Gradle's tasks

build.gradle

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

Output of **gradle -q count**

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

16.5. Task dependencies

As you probably have guessed, you can declare tasks that depend on other tasks.

Example 16.6. Declaration of task that depends on other task

build.gradle

```
task hello {
    doLast {
        println 'Hello world!'
    }
}
task intro(dependsOn: hello) {
    doLast {
        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 16.7. Lazy dependsOn - the other task does not exist (yet)

build.gradle

```
task taskX(dependsOn: 'taskY') {
    doLast {
        println 'taskX'
    }
}
task taskY {
    doLast {
        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 19.4, “Adding dependencies to a task”.

Please notice that you can't use shortcut notation (see Section 16.8, “Shortcut notations”) when referring to a task that is not yet defined.

16.6. 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 16.8. Dynamic creation of a task

build.gradle

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

Output of **gradle -q task1**

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

16.7. Manipulating existing tasks

Once tasks are created they can be accessed via an [*API*](#). For instance, you could use this to dynamically add dependencies to a task, at runtime. Ant doesn't allow anything like this.

Example 16.9. Accessing a task via API - adding a dependency

build.gradle

```
4.times { counter ->
    task "task$counter" {
        doLast {
            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 16.10. Accessing a task via API - adding behaviour

build.gradle

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

16.8. Shortcut notations

There is a convenient notation for accessing an *existing* task. Each task is available as a property of the build script:

Example 16.11. Accessing task as a property of the build script

build.gradle

```
task hello {
    doLast {
        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 tasks provided by the plugins, like the `compile` task.

16.9. Extra task properties

You can add your own properties to a task. To add a property named `myProperty`, set `ext.myProperty` to an initial value. From that point on, the property can be read and set like a predefined task property.

Example 16.12. Adding extra properties to a task

build.gradle

```
task myTask {
    ext.myProperty = "myValue"
}

task printTaskProperties {
    doLast {
        println myTask.myProperty
    }
}
```

Output of **gradle -q printTaskProperties**

```
> gradle -q printTaskProperties
myValue
```

Extra properties aren't limited to tasks. You can read more about them in Section 18.4.2, “Extra properties”.

16.10. Using Ant Tasks

Ant tasks are first-class citizens in Gradle. Gradle provides excellent integration for Ant tasks by simply 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. From the example below, you can learn how to execute Ant tasks and how to access Ant properties:

Example 16.13. Using AntBuilder to execute ant.loadfile target

build.gradle

```
task loadfile {
    doLast {
        def files = file('../antLoadfileResources').listFiles().sort()
        files.each { File file ->
            if (file.isFile()) {
                ant.loadfile(srcFile: file, property: file.name)
                println " *** $file.name ***"
                println "${ant.properties[file.name]}"
            }
        }
    }
}
```

Output of **gradle -q loadfile**

```
> gradle -q loadfile
*** agile.manifesto.txt ***
Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan
*** gradle.manifesto.txt ***
Make the impossible possible, make the possible easy and make the easy elegant.
(inspired by Moshe Feldenkrais)
```

There is lots more you can do with Ant in your build scripts. You can find out more in Chapter 21, *Using Ant from Gradle*.

16.11. 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 16.14. Using methods to organize your build logic

build.gradle

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

task loadfile {
    doLast {
        fileList('../antLoadfileResources').each { File file ->
            ant.loadfile(srcFile: file, property: file.name)
            println "I'm fond of $file.name"
        }
    }
}

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

Output of **gradle -q loadfile**

```
> gradle -q loadfile
I'm fond of agile.manifesto.txt
I'm fond of gradle.manifesto.txt
```

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 43, *Organizing Build Logic*.

16.12. Default tasks

Gradle allows you to define one or more default tasks that are executed if no other tasks are specified.

Example 16.15. Defining a default task

build.gradle

```
defaultTasks 'clean', 'run'

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

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

task other {
    doLast {
        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).

16.13. Configure by DAG

As we later describe in full detail (see Chapter 22, *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 among the tasks to be executed. Depending on this, you can assign different values to some variables.

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

Example 16.16. Different outcomes of build depending on chosen tasks

build.gradle

```
task distribution {
    doLast {
        println "We build the zip with version=$version"
    }
}

task release(dependsOn: 'distribution') {
    doLast {
        println 'We release now'
    }
}

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

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 `whenReady` affects the release task *before* the release task is executed. This works even when the release task is not the *primary* task (i.e., the task passed to the **gradle** command).

16.14. Where to next?

In this chapter, we have had a first look at tasks. But this is not the end of the story for tasks. If you want to jump into more of the details, have a look at Chapter 19, *More about Tasks*.

Otherwise, continue on to the tutorials in Chapter 46, *Java Quickstart* and Chapter 7, *Dependency Management Basics*.

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

17

Build Init Plugin

The Build Init plugin is currently incubating. Please be aware that the DSL and other configuration may change in later Gradle versions.

The Gradle Build Init plugin can be used to bootstrap the process of creating a new Gradle build. It supports creating brand new projects of different types as well as converting existing builds (e.g. An Apache Maven build) to be Gradle builds.

Gradle plugins typically need to be *applied* to a project before they can be used (see Section 27.3, “Using plugins”). The Build Init plugin is an automatically applied plugin, which means you do not need to apply it explicitly. To use the plugin, simply execute the task named `init` where you would like to create the Gradle build. There is no need to create a “stub” `build.gradle` file in order to apply the plugin.

It also leverages the `wrapper` task from the Wrapper plugin (see Chapter 23, *Wrapper Plugin*), which means that the Gradle Wrapper will also be installed into the project.

17.1. Tasks

The plugin adds the following tasks to the project:

Table 17.1. Build Init plugin - tasks

Task name	Depends on	Type	Description
<code>init</code>	<code>wrapper</code>	<code>InitBuild</code>	Generates a Gradle project.
<code>wrapper</code>	-	<code>Wrapper</code>	Generates Gradle wrapper files.

17.2. What to set up

The `init` supports different build setup *types*. The type is specified by supplying a `--type` argument value. For example, to create a Java library project simply execute: `gradle init --type java-library`.

If a `--type` parameter is not supplied, Gradle will attempt to infer the type from the environment. For example, it will infer a type value of “pom” if it finds a `pom.xml` to convert to a Gradle build.

If the type could not be inferred, the type “basic” will be used.

All build setup types include the setup of the Gradle Wrapper.

17.3. Build init types

As this plugin is currently incubating, only a few build init types are currently supported. More types will be added in future Gradle releases.

17.3.1. “pom” (Maven conversion)

The “pom” type can be used to convert an Apache Maven build to a Gradle build. This works by converting the POM to one or more Gradle files. It is only able to be used if there is a valid “pom.xml” file in the directory that the `init` task is invoked in or, if invoked via the “-p” cmdline option, in the specified project directory. This “pom” type will be automatically inferred if such a file exists.

The Maven conversion implementation was inspired by the maven2gradle tool that was originally developed by Gradle community members.

The conversion process has the following features:

- Uses effective POM and effective settings (support for POM inheritance, dependency management, properties)
- Supports both single module and multimodule projects
- Supports custom module names (that differ from directory names)
- Generates general metadata - id, description and version
- Applies maven, java and war plugins (as needed)
- Supports packaging war projects as jars if needed
- Generates dependencies (both external and inter-module)
- Generates download repositories (inc. local Maven repository)
- Adjusts Java compiler settings
- Supports packaging of sources and tests
- Supports TestNG runner
- Generates global exclusions from Maven enforcer plugin settings

17.3.2. “java-application”

The “java-application” build init type is not inferable. It must be explicitly specified.

It has the following features:

- Uses the “application” plugin to produce a command-line application implemented using Java
- Uses the “jcenter” dependency repository
- Uses JUnit for testing
- Has directories in the conventional locations for source code
- Contains a sample class and unit test, if there are no existing source or test files

Alternative test framework can be specified by supplying a `--test-framework` argument value. To use a different test framework, execute one of the following commands:

- `gradle init --type java-application --test-framework spock`: Uses Spock for testing instead of JUnit
- `gradle init --type java-application --test-framework testng`: Uses TestNG for testing instead of JUnit

17.3.3. “java-library”

The “java-library” build init type is not inferable. It must be explicitly specified.

It has the following features:

- Uses the “java” plugin to produce a library Jar
- Uses the “jcenter” dependency repository
- Uses JUnit for testing
- Has directories in the conventional locations for source code
- Contains a sample class and unit test, if there are no existing source or test files

Alternative test framework can be specified by supplying a `--test-framework` argument value. To use a different test framework, execute one of the following commands:

- `gradle init --type java-library --test-framework spock`: Uses Spock for testing instead of JUnit
- `gradle init --type java-library --test-framework testng`: Uses TestNG for testing instead of JUnit

17.3.4. “scala-library”

The “scala-library” build init type is not inferable. It must be explicitly specified.

It has the following features:

- Uses the “scala” plugin to produce a library Jar
- Uses the “jcenter” dependency repository
- Uses Scala 2.10
- Uses ScalaTest for testing
- Has directories in the conventional locations for source code
- Contains a sample scala class and an associated ScalaTest test suite, if there are no existing source or test files
- Uses the Zinc Scala compiler by default

17.3.5. “groovy-library”

The “groovy-library” build init type is not inferable. It must be explicitly specified.

It has the following features:

- Uses the “groovy” plugin to produce a library Jar
- Uses the “jcenter” dependency repository
- Uses Groovy 2.x
- Uses Spock testing framework for testing
- Has directories in the conventional locations for source code
- Contains a sample Groovy class and an associated Spock specification, if there are no existing source or test files

17.3.6. “basic”

The “basic” build init type is useful for creating a fresh new Gradle project. It creates a sample `build.gradle` file, with comments and links to help get started.

This type is used when no type was explicitly specified, and no type could be inferred.

18

Writing Build Scripts

This chapter looks at some of the details of writing a build script.

18.1. The Gradle build language

Gradle provides a *domain specific language*, or DSL, for describing builds. This build language is based on Groovy, with some additions to make it easier to describe a build.

A build script can contain any Groovy language element. ^[6] Gradle assumes that each build script is encoded using UTF-8.

18.2. The Project API

In the tutorial in Chapter 46, *Java Quickstart* we used, for example, the `apply()` method. Where does this method come from? We said earlier that the build script defines a project in Gradle. For each project in the build, Gradle creates an object of type `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.

Getting help writing build scripts

Don't forget that your build script is simply Groovy code that drives the Gradle API. And the `Project` interface is your starting point for accessing everything in the Gradle API. So, if you're wondering what 'tags' are available in your build script, you can start with the documentation for the `Project` interface.

Example 18.1. Accessing property of the Project object

build.gradle

```
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, would you need to use the `project` property.

18.2.1. 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 18.1. 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 project directory.
<code>path</code>	<code>String</code>	The absolute path of the project.
<code>description</code>	<code>String</code>	A description for the project.
<code>projectDir</code>	<code>File</code>	The directory containing the build script.
<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

18.3. The Script API

When Gradle executes a script, it compiles the script into a class which implements `Script`. This means that all of the properties and methods declared by the `Script` interface are available in your script.

18.4. Declaring variables

There are two kinds of variables that can be declared in a build script: local variables and extra properties.

18.4.1. Local variables

Local variables are declared with the `def` keyword. They are only visible in the scope where they have been declared. Local variables are a feature of the underlying Groovy language.

Example 18.2. Using local variables

build.gradle

```
def dest = "dest"

task copy(type: Copy) {
    from "source"
    into dest
}
```

18.4.2. Extra properties

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

Example 18.3. Using extra properties

build.gradle

```
apply plugin: "java"

ext {
    springVersion = "3.1.0.RELEASE"
    emailNotification = "build@master.org"
}

sourceSets.all { ext.purpose = null }

sourceSets {
    main {
        purpose = "production"
    }
    test {
        purpose = "test"
    }
    plugin {
        purpose = "production"
    }
}

task printProperties {
    doLast {
        println springVersion
        println emailNotification
        sourceSets.matching { it.purpose == "production" }.each { println it.name }
    }
}
```

Output of **gradle -q printProperties**

```
> gradle -q printProperties
3.1.0.RELEASE
build@master.org
main
plugin
```

In this example, an `ext` block adds two extra properties to the `project` object. Additionally, a property named `purpose` is added to each source set by setting `ext.purpose` to `null` (`null` is a permissible value). Once the properties have been added, they can be read and set like predefined properties.

By requiring special syntax for adding a property, Gradle can fail fast when an attempt is made to set a (predefined or extra) property but the property is misspelled or does not exist. Extra properties can be accessed from anywhere their owning object can be accessed, giving them a wider scope than local variables. Extra properties on a project are visible from its subprojects.

For further details on extra properties and their API, see the `ExtraPropertiesExtension` class in the API documentation.

18.5. Configuring arbitrary objects

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

Example 18.4. Configuring arbitrary objects

build.gradle

```
task configure {
    doLast {
        def 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
```

18.6. Configuring arbitrary objects using an external script

You can also configure arbitrary objects using an external script.

Example 18.5. Configuring arbitrary objects using a script

build.gradle

```
task configure {
    doLast {
        def pos = new java.text.FieldPosition(10)
        // Apply the script
        apply from: 'other.gradle', to: pos
        println pos.beginIndex
        println pos.endIndex
    }
}
```

other.gradle

```
// Set properties.
beginIndex = 1
endIndex = 5
```

Output of **gradle -q configure**

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

18.7. Some Groovy basics

The Groovy language provides plenty of features for creating DSLs, and the Gradle build language takes advantage of these. Understanding how the build language works will help you when you write your build script, and in particular, when you start to write custom plugins and tasks.

18.7.1. Groovy JDK

Groovy adds lots of useful methods to the standard Java classes. For example, `Iterable` gets an `each` method, which iterates over the elements of the `Iterable`:

Example 18.6. Groovy JDK methods

build.gradle

```
// Iterable gets an each() method
configurations.runtime.each { File f -> println f }
```

Have a look at <http://groovy-lang.org/gdk.html> for more details.

18.7.2. Property accessors

Groovy automatically converts a property reference into a call to the appropriate getter or setter method.

Example 18.7. Property accessors

build.gradle

```
// Using a getter method
println project.buildDir
println getProject().getBuildDir()

// Using a setter method
project.buildDir = 'target'
getProject().setBuildDir('target')
```

18.7.3. Optional parentheses on method calls

Parentheses are optional for method calls.

Example 18.8. Method call without parentheses

build.gradle

```
test.systemProperty 'some.prop', 'value'
test.systemProperty('some.prop', 'value')
```

18.7.4. List and map literals

Groovy provides some shortcuts for defining List and Map instances. Both kinds of literals are straightforward, but map literals have some interesting twists.

For instance, the “apply” method (where you typically apply plugins) actually takes a map parameter. However, when you have a line like “apply plugin: 'java'”, you aren't actually using a map literal, you're actually using “named parameters”, which have almost exactly the same syntax as a map literal (without the wrapping brackets). That named parameter list gets converted to a map when the method is called, but it doesn't start out as a map.

Example 18.9. List and map literals

build.gradle

```
// List literal
test.includes = ['org/gradle/api/**', 'org/gradle/internal/**']

List<String> list = new ArrayList<String>()
list.add('org/gradle/api/**')
list.add('org/gradle/internal/**')
test.includes = list

// Map literal.
Map<String, String> map = [key1:'value1', key2: 'value2']

// Groovy will coerce named arguments
// into a single map argument
apply plugin: 'java'
```

18.7.5. Closures as the last parameter in a method

The Gradle DSL uses closures in many places. You can find out more about closures [here](#). When the last parameter of a method is a closure, you can place the closure after the method call:

Example 18.10. Closure as method parameter

build.gradle

```
repositories {
    println "in a closure"
}
repositories() { println "in a closure" }
repositories({ println "in a closure" })
```

18.7.6. Closure delegate

Each closure has a `delegate` object, which Groovy uses to look up variable and method references which are not local variables or parameters of the closure. Gradle uses this for [configuration closures](#), where the `delegate` object is set to the object to be configured.

Example 18.11. Closure delegates

build.gradle

```
dependencies {
    assert delegate == project.dependencies
    testCompile('junit:junit:4.12')
    delegate.testCompile('junit:junit:4.12')
}
```

18.8. Default imports

To make build scripts more concise, Gradle automatically adds a set of import statements to the Gradle scripts. This means that instead of using `throw new org.gradle.api.tasks.StopExecutionException()` you can just type `throw new StopExecutionException()` instead.

Listed below are the imports added to each script:

Figure 18.1. gradle-imports

```
import org.gradle.*
import org.gradle.api.*
import org.gradle.api.artifacts.*
import org.gradle.api.artifacts.cache.*
import org.gradle.api.artifacts.component.*
import org.gradle.api.artifacts.dsl.*
import org.gradle.api.artifacts.ivy.*
import org.gradle.api.artifacts.maven.*
import org.gradle.api.artifacts.query.*
import org.gradle.api.artifacts.repositories.*
```

```
import org.gradle.api.artifacts.result.*
import org.gradle.api.artifacts.transform.*
import org.gradle.api.attributes.*
import org.gradle.api.component.*
import org.gradle.api.credentials.*
import org.gradle.api.distribution.*
import org.gradle.api.distribution.plugins.*
import org.gradle.api.dsl.*
import org.gradle.api.execution.*
import org.gradle.api.file.*
import org.gradle.api.initialization.*
import org.gradle.api.initialization.dsl.*
import org.gradle.api.invocation.*
import org.gradle.api.java.archives.*
import org.gradle.api.logging.*
import org.gradle.api.logging.configuration.*
import org.gradle.api.plugins.*
import org.gradle.api.plugins.announce.*
import org.gradle.api.plugins.antlr.*
import org.gradle.api.plugins.buildcomparison.gradle.*
import org.gradle.api.plugins.jetty.*
import org.gradle.api.plugins.osgi.*
import org.gradle.api.plugins.quality.*
import org.gradle.api.plugins.scala.*
import org.gradle.api.publish.*
import org.gradle.api.publish.ivy.*
import org.gradle.api.publish.ivy.plugins.*
import org.gradle.api.publish.ivy.tasks.*
import org.gradle.api.publish.maven.*
import org.gradle.api.publish.maven.plugins.*
import org.gradle.api.publish.maven.tasks.*
import org.gradle.api.publish.plugins.*
import org.gradle.api.reporting.*
import org.gradle.api.reporting.components.*
import org.gradle.api.reporting.dependencies.*
import org.gradle.api.reporting.dependents.*
import org.gradle.api.reporting.model.*
import org.gradle.api.reporting.plugins.*
import org.gradle.api.resources.*
import org.gradle.api.specs.*
import org.gradle.api.tasks.*
import org.gradle.api.tasks.ant.*
import org.gradle.api.tasks.application.*
import org.gradle.api.tasks.bundling.*
import org.gradle.api.tasks.compile.*
import org.gradle.api.tasks.diagnostics.*
import org.gradle.api.tasks.incremental.*
import org.gradle.api.tasks.javadoc.*
import org.gradle.api.tasks.scala.*
import org.gradle.api.tasks.testing.*
import org.gradle.api.tasks.testing.junit.*
import org.gradle.api.tasks.testing.testng.*
import org.gradle.api.tasks.util.*
import org.gradle.api.tasks.wrapper.*
import org.gradle.authentication.*
import org.gradle.authentication.aws.*
import org.gradle.authentication.http.*
import org.gradle.buildinit.plugins.*
import org.gradle.buildinit.tasks.*
import org.gradle.caching.*
import org.gradle.external.javadoc.*
```

```
import org.gradle.ide.visualstudio.*
import org.gradle.ide.visualstudio.plugins.*
import org.gradle.ide.visualstudio.tasks.*
import org.gradle.ivy.*
import org.gradle.jvm.*
import org.gradle.jvm.application.scripts.*
import org.gradle.jvm.application.tasks.*
import org.gradle.jvm.platform.*
import org.gradle.jvm.plugins.*
import org.gradle.jvm.tasks.*
import org.gradle.jvm.tasks.api.*
import org.gradle.jvm.test.*
import org.gradle.jvm.toolchain.*
import org.gradle.language.assembler.*
import org.gradle.language.assembler.plugins.*
import org.gradle.language.assembler.tasks.*
import org.gradle.language.base.*
import org.gradle.language.base.artifact.*
import org.gradle.language.base.plugins.*
import org.gradle.language.base.sources.*
import org.gradle.language.c.*
import org.gradle.language.c.plugins.*
import org.gradle.language.c.tasks.*
import org.gradle.language.coffeescript.*
import org.gradle.language.cpp.*
import org.gradle.language.cpp.plugins.*
import org.gradle.language.cpp.tasks.*
import org.gradle.language.java.*
import org.gradle.language.java.artifact.*
import org.gradle.language.java.plugins.*
import org.gradle.language.java.tasks.*
import org.gradle.language.javascript.*
import org.gradle.language.jvm.*
import org.gradle.language.jvm.plugins.*
import org.gradle.language.jvm.tasks.*
import org.gradle.language.nativeplatform.*
import org.gradle.language.nativeplatform.tasks.*
import org.gradle.language.objectivec.*
import org.gradle.language.objectivec.plugins.*
import org.gradle.language.objectivec.tasks.*
import org.gradle.language.objectivec.cpp.*
import org.gradle.language.objectivec.cpp.plugins.*
import org.gradle.language.objectivec.cpp.tasks.*
import org.gradle.language.rc.*
import org.gradle.language.rc.plugins.*
import org.gradle.language.rc.tasks.*
import org.gradle.language.routes.*
import org.gradle.language.scala.*
import org.gradle.language.scala.plugins.*
import org.gradle.language.scala.tasks.*
import org.gradle.language.scala.toolchain.*
import org.gradle.language.twirl.*
import org.gradle.maven.*
import org.gradle.model.*
import org.gradle.nativeplatform.*
import org.gradle.nativeplatform.platform.*
import org.gradle.nativeplatform.plugins.*
import org.gradle.nativeplatform.tasks.*
import org.gradle.nativeplatform.test.*
import org.gradle.nativeplatform.test.cunit.*
import org.gradle.nativeplatform.test.cunit.plugins.*
```

```
import org.gradle.nativeplatform.test.cunit.tasks.*
import org.gradle.nativeplatform.test.googletest.*
import org.gradle.nativeplatform.test.googletest.plugins.*
import org.gradle.nativeplatform.test.plugins.*
import org.gradle.nativeplatform.test.tasks.*
import org.gradle.nativeplatform.toolchain.*
import org.gradle.nativeplatform.toolchain.plugins.*
import org.gradle.platform.base.*
import org.gradle.platform.base.binary.*
import org.gradle.platform.base.component.*
import org.gradle.platform.base.plugins.*
import org.gradle.play.*
import org.gradle.play.distribution.*
import org.gradle.play.platform.*
import org.gradle.play.plugins.*
import org.gradle.play.plugins.ide.*
import org.gradle.play.tasks.*
import org.gradle.play.toolchain.*
import org.gradle.plugin.devel.*
import org.gradle.plugin.devel.plugins.*
import org.gradle.plugin.devel.tasks.*
import org.gradle.plugin.repository.*
import org.gradle.plugin.use.*
import org.gradle.plugins.ear.*
import org.gradle.plugins.ear.descriptor.*
import org.gradle.plugins.ide.api.*
import org.gradle.plugins.ide.eclipse.*
import org.gradle.plugins.ide.idea.*
import org.gradle.plugins.javascript.base.*
import org.gradle.plugins.javascript.coffeescript.*
import org.gradle.plugins.javascript.envjs.*
import org.gradle.plugins.javascript.envjs.browser.*
import org.gradle.plugins.javascript.envjs.http.*
import org.gradle.plugins.javascript.envjs.http.simple.*
import org.gradle.plugins.javascript.jshint.*
import org.gradle.plugins.javascript.rhino.*
import org.gradle.plugins.javascript.rhino.worker.*
import org.gradle.plugins.signing.*
import org.gradle.plugins.signing.signatory.*
import org.gradle.plugins.signing.signatory.pgp.*
import org.gradle.plugins.signing.type.*
import org.gradle.plugins.signing.type.pgp.*
import org.gradle.process.*
import org.gradle.process.daemon.*
import org.gradle.testing.base.*
import org.gradle.testing.base.plugins.*
import org.gradle.testing.jacoco.plugins.*
import org.gradle.testing.jacoco.tasks.*
import org.gradle.testing.jacoco.tasks.rules.*
```

```
import org.gradle.testkit.runner.*  
import org.gradle.util.*
```

[6] Any language element except for statement labels.

19

More about Tasks

In the introductory tutorial (Chapter 16, *Build Script Basics*) you learned how to create simple tasks. You also learned how to add additional behavior to these tasks later on, and you 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*, which are tasks that have their own properties and methods. This is really different from what you are used to with Ant targets. Such enhanced tasks are either provided by you or built into Gradle.

19.1. Defining tasks

We have already seen how to define tasks using a keyword style in Chapter 16, *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 19.1. Defining tasks

build.gradle

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

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

You can also use strings for the task names:

Example 19.2. Defining tasks - using strings for task names

build.gradle

```
task('hello') {
    doLast {
        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 19.3. Defining tasks with alternative syntax

build.gradle

```
tasks.create(name: 'hello') {
    doLast {
        println "hello"
    }
}

tasks.create(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 `create` method.

19.2. 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 of doing this. Firstly, each task is available as a property of the project, using the task name as the property name:

Example 19.4. Accessing tasks as properties

build.gradle

```
task hello

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

Tasks are also available through the `tasks` collection.

Example 19.5. 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 19.6. 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.

19.3. 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:

Example 19.7. 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.

Just to be clear, realize that the name of this task is “myCopy”, but it is of *type* “Copy”. You can have multiple tasks of the same *type*, but with different names. You'll find this gives you a lot of power to

implement cross-cutting concerns across all tasks of a particular type.

Example 19.8. 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 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 another way of configuring a task. It also preserves the context and it is arguably the most readable. It is usually our favorite.

Example 19.9. 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, not when the task executes.

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

Example 19.10. Defining a task with closure

build.gradle

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

Don't forget about the build phases

A task has both configuration and actions. When using the `<<`, you are simply using a shortcut to define an action. Code defined in the configuration

19.4. Adding dependencies to a task

There are several ways you can define the dependencies of a task. In Section 16.5, “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. The following is an example which adds a dependency from `projectA:taskX` to `projectB:taskY`:

section of your task will get executed during the configuration phase of the build regardless of what task was targeted. See Chapter 22, *The Build Lifecycle* for more details about the build lifecycle.

Example 19.11. Adding dependency on task from another project

build.gradle

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

project('projectB') {
    task taskY {
        doLast {
            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 19.12. Adding dependency using task object

build.gradle

```
task taskX {
    doLast {
        println 'taskX'
    }
}

task taskY {
    doLast {
        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 19.13. Adding dependency using closure

build.gradle

```
task taskX {
    doLast {
        println 'taskX'
    }
}

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

task lib1 {
    doLast {
        println 'lib1'
    }
}

task lib2 {
    doLast {
        println 'lib2'
    }
}

task notALib {
    doLast {
        println 'notALib'
    }
}
```

Output of `gradle -q taskX`

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

For more information about task dependencies, see the Task API.

19.5. Ordering tasks

Task ordering is an incubating feature. Please be aware that this feature may change in later Gradle versions.

In some cases it is useful to control the *ordering* in which 2 tasks will execute, without introducing an explicit dependency between those tasks. The primary difference between a task *ordering* and a task *dependency* is that an ordering rule does not influence which tasks will be executed, only the order in which they will be executed.

Task ordering can be useful in a number of scenarios:

- Enforce sequential ordering of tasks: e.g. 'build' never runs before 'clean'.
- Run build validations early in the build: e.g. validate I have the correct credentials before starting the work for a release build.
- Get feedback faster by running quick verification tasks before long verification tasks: e.g. unit tests should run before integration tests.
- A task that aggregates the results of all tasks of a particular type: e.g. test report task combines the outputs of all executed test tasks.

There are two ordering rules available: “*must run after*” and “*should run after*”.

When you use the “must run after” ordering rule you specify that `taskB` must always run after `taskA`, whenever both `taskA` and `taskB` will be run. This is expressed as `taskB.mustRunAfter(taskA)`. The “should run after” ordering rule is similar but less strict as it will be ignored in two situations. Firstly if using that rule introduces an ordering cycle. Secondly when using parallel execution and all dependencies of a task have been satisfied apart from the “should run after” task, then this task will be run regardless of whether its “should run after” dependencies have been run or not. You should use “should run after” where the ordering is helpful but not strictly required.

With these rules present it is still possible to execute `taskA` without `taskB` and vice-versa.

Example 19.14. Adding a 'must run after' task ordering

build.gradle

```
task taskX {
    doLast {
        println 'taskX'
    }
}
task taskY {
    doLast {
        println 'taskY'
    }
}
taskY.mustRunAfter taskX
```

Output of `gradle -q taskY taskX`

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


Example 19.15. Adding a 'should run after' task ordering

build.gradle

```
task taskX {
    doLast {
        println 'taskX'
    }
}
task taskY {
    doLast {
        println 'taskY'
    }
}
taskY.shouldRunAfter taskX
```

Output of **gradle -q taskY taskX**

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

In the examples above, it is still possible to execute `taskY` without causing `taskX` to run:

Example 19.16. Task ordering does not imply task execution

Output of **gradle -q taskY**

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

To specify a “must run after” or “should run after” ordering between 2 tasks, you use the `Task.mustRunAfter(java.lang.Object[])` and `Task.shouldRunAfter(java.lang.Object[])` methods. These methods accept a task instance, a task name or any other input accepted by `Task.dependsOn(java.lang.Object[])`.

Note that “`B.mustRunAfter(A)`” or “`B.shouldRunAfter(A)`” does not imply any execution dependency between the tasks:

- It is possible to execute tasks A and B independently. The ordering rule only has an effect when both tasks are scheduled for execution.
- When run with `--continue`, it is possible for B to execute in the event that A fails.

As mentioned before, the “should run after” ordering rule will be ignored if it introduces an ordering cycle:

Example 19.17. A 'should run after' task ordering is ignored if it introduces an ordering cycle

build.gradle

```
task taskX {
    doLast {
        println 'taskX'
    }
}
task taskY {
    doLast {
        println 'taskY'
    }
}
task taskZ {
    doLast {
        println 'taskZ'
    }
}
taskX.dependsOn taskY
taskY.dependsOn taskZ
taskZ.shouldRunAfter taskX
```

Output of **gradle -q taskX**

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

19.6. Adding a description to a task

You can add a description to your task. This description is displayed when executing **gradle tasks**.

Example 19.18. 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')
}
```

19.7. 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 19.19. Overwriting a task

build.gradle

```
task copy(type: Copy)

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

Output of **gradle -q copy**

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

This will replace a task of type `Copy` with the task you've defined, because it uses the same name. When you define the new task, you have to set the `overwrite` property to `true`. Otherwise Gradle throws an exception, saying that a task with that name already exists.

19.8. Skipping tasks

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

19.8.1. 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 19.20. Skipping a task using a predicate

build.gradle

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

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

Output of **gradle hello -PskipHello**

```
> gradle hello -PskipHello
:hello SKIPPED
```

```
BUILD SUCCESSFUL
```

```
Total time: 1 secs
```

19.8.2. Using StopExecutionException

If the logic for skipping a task can't be expressed with a 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 19.21. Skipping tasks with `StopExecutionException`

build.gradle

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

compile.doFirst {
    // Here you would put arbitrary conditions in real life.
    // But this is used in an integration test so we want defined behavior.
    if (true) { throw new StopExecutionException() }
}

task myTask(dependsOn: 'compile') {
    doLast {
        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. ^[7]

19.8.3. Enabling and disabling tasks

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

Example 19.22. Enabling and disabling tasks

build.gradle

```
task disableMe {
    doLast {
        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
```

19.9. Up-to-date checks (AKA Incremental Build)

An important part of any build tool is the ability to avoid doing work that has already been done. Consider the process of compilation. Once your source files have been compiled, there should be no need to recompile them unless something has changed that affects the output, such as the modification of a source file or the removal of an output file. And compilation can take a significant amount of time, so skipping the step when it's not needed saves a lot of time.

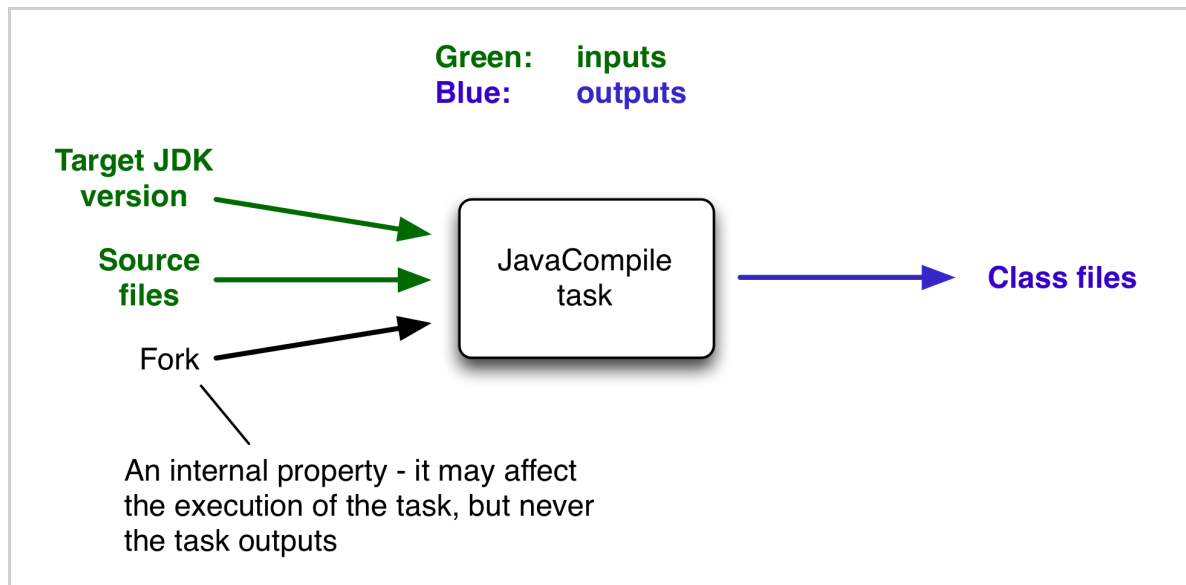
Gradle supports this behavior out of the box through a feature it calls incremental build. You have almost certainly already seen it in action: it's active nearly^[8] every time the UP-TO-DATE text appears next to the name of a task when you run a build.

How does incremental build work? And what does it take to make use of it in your own tasks? Let's take a look.

19.9.1. Task inputs and outputs

In the most common case, a task takes some inputs and generates some outputs. If we use the compilation example from earlier, we can see that the source files are the inputs and, in the case of Java, the generated class files are the outputs. Other inputs might include things like whether debug information should be included.

Figure 19.1. Example task inputs and outputs



An important characteristic of an input is that it affects one or more outputs, as you can see from the previous figure. Different bytecode is generated depending on the content of the source files and the minimum version of the Java runtime you want to run the code on. That makes them task inputs. But whether the compilation runs in a forked process or not, determined by the `fork` property, has no impact on what bytecode gets generated. In Gradle terminology, `fork` is just an internal task property.

As part of incremental build, Gradle tests whether any of the task inputs or outputs have changed since the last build. If they haven't, Gradle can consider the task up to date and therefore skip executing its actions. Also note that incremental build won't work unless a task has at least one task output, although tasks usually have at least one input as well.

What this means for build authors is simple: you need to tell Gradle which task properties are inputs and which are outputs. If a task property affects the output, be sure to register it as an input, otherwise the task will be considered up to date when it's not. Conversely, don't register properties as inputs if they don't affect the output, otherwise the task will potentially execute when it doesn't need to. Also be careful of non-deterministic tasks that may generate different output for exactly the same inputs: these should not be configured for incremental build as the up-to-date checks won't work.

Let's now look at how you can register task properties as inputs and outputs.

Custom task types

If you're implementing a custom task as a class, then it takes just two steps to make it work with incremental build:

1. Create typed fields or properties (via getter methods) for each of your task inputs and outputs
2. Add the appropriate annotation to each of those fields or getter methods

Gradle supports three main categories of inputs and outputs:

- Simple values

Things like strings and numbers. More generally, a simple value can have any type that implements `Serializable`

- Filesystem types

These consist of the standard `File` class but also derivatives of Gradle's `FileCollection` type and anything else that can be passed to either the `Project.file(java.lang.Object)` method - for single file/directory properties - or the `Project.files(java.lang.Object[])` method.

- Nested values

Custom types that don't conform to the other two categories but have their own properties that are inputs or outputs. In effect, the task inputs or outputs are nested inside these custom types.

As an example, imagine you have a task that processes templates of varying types, such as FreeMarker, Velocity, Moustache, etc. It takes template source files and combines them with some model data to generate populated versions of the template files.

This task will have three inputs and one output:

- Template source files
- Model data
- Template engine
- Where the output files are written

When you're writing a custom task class, it's easy to register properties as inputs or outputs via annotations. To demonstrate, here is a skeleton task implementation with some suitable inputs and outputs, along with their annotations:

Example 19.23. Custom task class

buildSrc/src/main/java/org/example/ProcessTemplates.java

```

package org.example;

import java.io.File;
import java.util.HashMap;
import org.gradle.api.*;
import org.gradle.api.file.*;
import org.gradle.api.tasks.*;

public class ProcessTemplates extends DefaultTask {
    private TemplateEngineType templateEngine;
    private FileCollection sourceFiles;
    private TemplateData templateData;
    private File outputDir;

    @Input
    public TemplateEngineType getTemplateEngine() {
        return this.templateEngine;
    }

    @InputFiles
    public FileCollection getSourceFiles() {
        return this.sourceFiles;
    }

    @Nested
    public TemplateData getTemplateData() {
        return this.templateData;
    }

    @OutputDirectory
    public File getOutputDir() { return this.outputDir; }

    // + setter methods for the above - assume we've defined them

    @TaskAction
    public void processTemplates() {
        // ...
    }
}

```

buildSrc/src/main/java/org/example/TemplateData.java


```

package org.example;

import java.util.HashMap;
import java.util.Map;
import org.gradle.api.tasks.Input;

public class TemplateData {
    private String name;
    private Map<String, String> variables;

    public TemplateData(String name, Map<String, String> variables) {
        this.name = name;
        this.variables = new HashMap<>(variables);
    }

    @Input
    public String getName() { return this.name; }

    @Input
    public Map<String, String> getVariables() {
        return this.variables;
    }
}

```

Output of **gradle processTemplates**

```

> gradle processTemplates
:processTemplates

```

BUILD SUCCESSFUL

Output of **gradle processTemplates**

```

> gradle processTemplates
:processTemplates UP-TO-DATE

```

BUILD SUCCESSFUL

There's plenty to talk about in this example, so let's work through each of the input and output properties in turn:

- **templateEngine**
Represents which engine to use when processing the source templates, e.g. FreeMarker, Velocity, etc. You could implement this as a string, but in this case we have gone for a custom enum as it provides greater type information and safety. Since enums implement `Serializable` automatically, we can treat this as a simple value and use the `@Input` annotation, just as we would with a `String` property.
- **sourceFiles**
The source templates that the task will be processing. Single files and collections of files need their own special annotations. In this case, we're dealing with a collection of input files and so we use the `@InputFile` annotation. You'll see more file-oriented annotations in a table later.
- **templateData**
For this example, we're using a custom class to represent the model data. However, it does not implement `Serializable`, so we can't use the `@Input` annotation. That's not a problem as the properties within `TemplateData` - a string and a hash map with serializable type parameters - are

serializable and can be annotated with `@Input`. We use `@Nested` on `templateData` to let Gradle know that this is a value with nested input properties.

- `outputDir`

The directory where the generated files go. As with input files, there are several annotations for output files and directories. A property representing a single directory requires `@OutputDirectory`. You'll learn about the others soon.

These annotated properties mean that Gradle will skip the task if none of the source files, template engine, model data or generated files have changed since the previous time Gradle executed the task. This will often save a significant amount of time. You can learn how Gradle detects changes later.

This example is particularly interesting because it works with collections of source files. What happens if only one source file changes? Does the task process all the source files again or just the modified one? That depends on the task implementation. If the latter, then the task itself is incremental, but that's a different feature to the one we're discussing here. Gradle does help task implementers with this via its incremental task input feature.

Now that you have seen some of the input and output annotations in practice, let's take a look at all the annotations available to you and when you should use them. The table below lists the available annotations and the corresponding property type you can use with each one.

Table 19.1. Incremental build property type annotations

Annotation	Expected property type	Description
------------	------------------------	-------------

@Input	Any serializable type	A simple input value
@InputFile	File*	A single input file (not directory)
@InputDirectory	File*	A single input directory (not file)
@InputFiles	Iterable<File>*	An iterable of input files and directories
@Classpath	Iterable<File>*	An iterable of input files and directories that represent a Java classpath. This allows the task to ignore irrelevant changes to the property, such as different names for the same files. It is similar to annotating the property @PathSensitive(RELATIVE) but it will ignore the names of JAR files directly added to the classpath, and it will consider changes in the order of the files as a change in the classpath.
<div> <p>To stay compatible with Gradle versions before 3.2, classpath properties should also be annotated with @InputFiles.</p> </div>		
@OutputFile	File*	A single output file (not directory)
@OutputDirectory	File*	A single output directory (not file)
@OutputFiles	Map<String, File> ** or Iterable<File> *	An iterable of output files (no directories)
@OutputDirectories	Map<String, File> ** or Iterable<File> *	An iterable of output directories (no files)
@Nested	Any custom type	A custom type that may not implement Serializable but does have at least one field or property marked with one of the annotations in this table. It could even be another @Nested.
@Console	Any type	Indicates that the property is neither an input nor an output. It simply affects the console output of the task in some way, such as increasing or decreasing the verbosity of the task.
@Internal	Any type	Indicates that the property is used internally but is neither an input nor an output.

** In fact, `File` can be any type accepted by `Project.file(java.lang.Object)` and `Iterable<File>` can be any type accepted by `Project.files(java.lang.Object[])`. This includes instances of `Callable`, such as closures, allowing for lazy evaluation of the property values. Be aware that the types `FileCollection` and `FileTree` are `Iterable<File>`s.*

*** Similar to the above, `File` can be any type accepted by `Project.file(java.lang.Object)`. The `Map` itself can be wrapped in `Callables`, such as closures.*

Table 19.2. Additional annotations used to further qualifying property type annotations

Annotation	Description
<code>@SkipWhenEmpty</code>	Used with <code>@InputFiles</code> or <code>@InputDirectory</code> to tell Gradle to skip the task if the corresponding files or directory are empty, along with all other input files declared with this annotation. Tasks that have been skipped due to all of their input files that were declared with this annotation being empty will result in a distinct “no source” outcome. For example, <code>NO-SOURCE</code> will be emitted in the console output.
<code>@Optional</code>	Used with any of the property type annotations listed in the <code>Optional</code> API documentation. This annotation disables validation checks on the corresponding property. See the section on validation for more details.
<code>@OrderSensitive</code>	Used with <code>@InputFiles</code> or <code>@InputDirectory</code> to tell Gradle that a change in the order of the files should mark the task out-of-date. <div> <p>This functionality is deprecated and will be removed in Gradle 4.0. For classpath properties, use <code>@Classpath</code> instead.</p> </div>
<code>@PathSensitive</code>	Used with any input file property to tell Gradle to only consider the given part of the file paths as important. For example, if a property is annotated with <code>@PathSensitive</code> , then moving the files around without changing their contents will not make the task out-of-date.

Annotations are inherited from all parent types including implemented interfaces. Property type annotations override any other property type annotation declared in a parent type. This way an `@InputFile` property can be turned into an `@InputDirectory` property in a child task type.

Annotations on a property declared in a type override similar annotations declared by the superclass and in any implemented interfaces. Superclass annotations take precedence over annotations declared in implemented interfaces.

The `Console` and `Internal` annotations in the table are special cases as they don’t declare either task

inputs or task outputs. So why use them? It's so that you can take advantage of the Java Gradle Plugin Developer to help you develop and publish your own plugins. This plugin checks whether any properties of your custom task classes lack an incremental build annotation. This protects you from forgetting to add an appropriate annotation during development.

Custom task classes are an easy way to bring your own build logic into the arena of incremental build, but you don't always have that option. That's why Gradle also provides an alternative API that can be used with any tasks, which we look at next.

Runtime API

When you don't have access to the source for a custom task class, there is no way to add any of the annotations we covered in the previous section. Fortunately, Gradle provides a runtime API for scenarios just like that. It can also be used for ad-hoc tasks, as you'll see next.

Using it for ad-hoc tasks

This runtime API is provided through a couple of aptly named properties that are available on every Gradle task:

- `Task.getInputs()` of type `TaskInputs`
- `Task.getOutputs()` of type `TaskOutputs`

These objects have methods that allow you to specify files, directories and values which constitute the task's inputs and outputs. In fact, the runtime API has almost feature parity with the annotations. All it lacks is validation of whether declared files are actually files and declared directories are directories. Nor will it create output directories if they don't exist. But that's it.

Let's take the template processing example from before and see how it would look as an ad-hoc task that uses the runtime API:

Example 19.24. Ad-hoc task

build.gradle

```
task processTemplatesAdHoc {
    inputs.property("engine", TemplateEngineType.FREEMARKER)
    inputs.files(fileTree("src/templates"))
    inputs.property("templateData.name", "docs")
    inputs.property("templateData.variables", [year: 2013])
    outputs.dir("$buildDir/genOutput2")

    doLast {
        // Process the templates here
    }
}
```

Output of **gradle processTemplatesAdHoc**

```
> gradle processTemplatesAdHoc
:processTemplatesAdHoc
```

```
BUILD SUCCESSFUL
```

As before, there's much to talk about. To begin with, you should really write a custom task class for this as it's a non-trivial implementation that has several configuration options. In this case, there are no task properties to store the root source folder, the location of the output directory or any of the other settings. That's deliberate to highlight the fact that the runtime API doesn't require the task to have any state. In terms of incremental build, the above ad-hoc task will behave the same as the custom task class.

All the input and output definitions are done through the methods on `inputs` and `outputs`, such as `property()`, `files()`, and `dir()`. Gradle performs up-to-date checks on the argument values to determine whether the task needs to run again or not. Each method corresponds to one of the incremental build annotations, for example `inputs.property()` maps to `@Input` and `outputs.dir()` maps to `@OutputDirectory`. The only difference is that the `file()`, `files()`, `dir()` and `dirs()` methods don't validate the type of file object at the given path (file or directory), unlike the annotations.

One notable difference between the runtime API and the annotations is the lack of a method that corresponds directly to `@Nested`. That's why the example uses two `property()` declarations for the template data, one for each `TemplateData` property. You should utilize the same technique when using the runtime API with nested values.

Using it for custom task types

Another type of example involves adding input and output definitions to instances of a custom task class that lacks the requisite annotations. For example, imagine that the `ProcessTemplates` task is provided by a plugin and that it's missing the incremental build annotations. In order to make up for that deficiency, you can use the runtime API:

Example 19.25. Using runtime API with custom task type

build.gradle

```
task processTemplatesRuntime(type: ProcessTemplatesNoAnnotations) {
    templateEngine = TemplateEngineType.FREEMARKER
    sourceFiles = fileTree("src/templates")
    templateData = new TemplateData("test", [year: 2014])
    outputDir = file("$buildDir/genOutput3")

    inputs.property("engine", templateEngine)
    inputs.files(sourceFiles)
    inputs.property("templateData.name", templateData.name)
    inputs.property("templateData.variables", templateData.variables)
    outputs.dir(outputDir)
}
```

Output of **gradle processTemplatesRuntime**

```
> gradle processTemplatesRuntime
:processTemplatesRuntime
```

BUILD SUCCESSFUL

Output of **gradle processTemplatesRuntime**

```
> gradle processTemplatesRuntime
:processTemplatesRuntime UP-TO-DATE
```

BUILD SUCCESSFUL

As you can see, we can both configure the tasks properties and use those properties as arguments to the incremental build runtime API. Using the runtime API like this is a little like using `doLast()` and `doFirst()` to attach extra actions to a task, except in this case we're attaching information about inputs and outputs. Note that if the task type is already using the incremental build annotations, the runtime API will add inputs and outputs rather than replace them.

Fine-grained configuration

The runtime API methods only allow you to declare your inputs and outputs in themselves. However, the file-oriented ones return a builder - of type `TaskInputFilePropertyBuilder` - that lets you provide additional information about those inputs and outputs.

You can learn about all the options provided by the builder in its API documentation, but we'll show you a simple example here to give you an idea of what you can do.

Let's say we don't want to run the `processTemplates` task if there are no source files, regardless of whether it's a clean build or not. After all, if there are no source files, there's nothing for the task to do. The builder allows us to configure this like so:

Example 19.26. Using skipWhenEmpty() via the runtime API

build.gradle

```
task processTemplatesRuntimeConf(type: ProcessTemplatesNoAnnotations) {  
    // ...  
    sourceFiles = fileTree("src/templates") {  
        include "**/*.fm"  
    }  
  
    inputs.files(sourceFiles).skipWhenEmpty()  
    // ...  
}
```

Output of **gradle clean processTemplatesRuntimeConf**

```
> gradle clean processTemplatesRuntimeConf  
:processTemplatesRuntimeConf NO-SOURCE  
  
BUILD SUCCESSFUL
```

The `TaskInputs.files()` method returns a builder that has a `skipWhenEmpty()` method. Invoking this method is equivalent to annotating to the property with `@SkipWhenEmpty`.

Prior to Gradle 3.0, you had to use the `TaskInputs.source()` and `TaskInputs.sourceDir()` methods to get the same behavior as with `skipWhenEmpty()`. These methods are now deprecated and should not be used with Gradle 3.0 and above.

Now that you have seen both the annotations and the runtime API, you may be wondering which API you should be using. Our recommendation is to use the annotations wherever possible, and it's sometimes worth creating a custom task class just so that you can make use of them. The runtime API is more for situations in which you can't use the annotations.

Important beneficial side effects

Once you declare a task's formal inputs and outputs, Gradle can then infer things about those properties. For example, if an input of one task is set to the output of another, that means the first task depends on the second, right? Gradle knows this and can act upon it.

We'll look at this feature next and also some other features that come from Gradle knowing things about inputs and outputs.

Inferred task dependencies

Consider an archive task that packages the output of the `processTemplates` task. A build author will see that the archive task obviously requires `processTemplates` to run first and so may add an explicit dependency. However, if you define the archive task like so:

Example 19.27. Inferred task dependency via task outputs

build.gradle

```
task packageFiles(type: Zip) {  
    from processTemplates.outputs  
}
```

Output of **gradle clean packageFiles**

```
> gradle clean packageFiles  
:processTemplates  
:packageFiles  
  
BUILD SUCCESSFUL
```

Gradle will automatically make `packageFiles` depend on `processTemplates`. It can do this because it's aware that one of the inputs of `packageFiles` requires the output of the `processTemplates` task. We call this an inferred task dependency.

The above example can also be written as

Example 19.28. Inferred task dependency via a task argument

build.gradle

```
task packageFiles2(type: Zip) {  
    from processTemplates  
}
```

Output of **gradle clean packageFiles2**

```
> gradle clean packageFiles2  
:processTemplates  
:packageFiles2  
  
BUILD SUCCESSFUL
```

This is because the `from()` method can accept a task object as an argument. Behind the scenes, `from()` uses the `project.files()` method to wrap the argument, which in turn exposes the task's formal outputs as a file collection. In other words, it's a special case!

Input and output validation

The incremental build annotations provide enough information for Gradle to perform some basic validation on the annotated properties. In particular, it does the following for each property before the task executes:

- `@InputFile` - verifies that the property has a value and that the path corresponds to a file (not a directory) that exists.
- `@InputDirectory` - same as for `@InputFile`, except the path must correspond to a directory.
- `@OutputDirectory` - verifies that the path doesn't match a file and also creates the directory if it doesn't already exist.

Such validation improves the robustness of the build, allowing you to identify issues related to inputs and

outputs quickly.

You will occasionally want to disable some of this validation, specifically when an input file may validly not exist. That's why Gradle provides the `@Optional` annotation: you use it to tell Gradle that a particular input is optional and therefore the build should not fail if the corresponding file or directory doesn't exist.

Continuous build

One last benefit of defining task inputs and outputs is continuous build. Since Gradle knows what files a task depends on, it can automatically run a task again if any of its inputs change. By activating continuous build when you run Gradle - through the `--continuous` or `-t` options - you will put Gradle into a state in which it continually checks for changes and executes the requested tasks when it encounters such changes.

You can find out more about this feature in Chapter 9, *Continuous build*.

19.9.2. Advanced techniques

Everything you've seen so far in this section will cover most of the use cases you'll encounter, but there are some scenarios that need special treatment. We'll present a few of those next with the appropriate solutions.

Adding your own cached input/output methods

Have you ever wondered how the `from()` method of the `Copy` task works? It's not annotated with `@InputFile` and yet any files passed to it are treated as formal inputs of the task. What's happening?

The implementation is quite simple and you can use the same technique for your own tasks to improve their APIs. Write your methods so that they add files directly to the appropriate annotated property. As an example, here's how to add a `sources()` method to the custom `ProcessTemplates` class we introduced earlier:

Example 19.29. Declaring a method to add task inputs

buildSrc/src/main/java/org/example/ProcessTemplates.java

```
public class ProcessTemplates extends DefaultTask {
    // ...
    private FileCollection sourceFiles = getProject().files();

    @SkipWhenEmpty
    @InputFiles
    @PathSensitive(PathSensitivity.NONE)
    public FileCollection getSourceFiles() {
        return this.sourceFiles;
    }

    public void sources(FileCollection sourceFiles) {
        this.sourceFiles = this.sourceFiles.plus(sourceFiles);
    }

    // ...
}
```

build.gradle

```
task processTemplates(type: ProcessTemplates) {
    templateEngine = TemplateEngineType.FREEMARKER
    templateData = new TemplateData("test", [year: 2012])
    outputDir = file("$buildDir/genOutput")

    sources fileTree("src/templates")
}
```

Output of **gradle processTemplates**

```
> gradle processTemplates
:processTemplates

BUILD SUCCESSFUL
```

In other words, as long as you add values and files to formal task inputs and outputs during the configuration phase, they will be treated as such regardless from where in the build you add them.

If we want to support tasks as arguments as well and treat their outputs as the inputs, we can use the `project.f:` method like so:

Example 19.30. Declaring a method to add a task as an input

buildSrc/src/main/java/org/example/ProcessTemplates.java

```
// ...
public void sources(Task inputTask) {
    this.sourceFiles = this.sourceFiles.plus(getProject().files(inputTask));
}
// ...
```

build.gradle

```
task copyTemplates(type: Copy) {
    into "$buildDir/tmp"
    from "src/templates"
}

task processTemplates2(type: ProcessTemplates) {
    // ...
    sources copyTemplates
}
```

Output of **gradle processTemplates2**

```
> gradle processTemplates2
:copyTemplates
:processTemplates2

BUILD SUCCESSFUL
```

This technique can make your custom task easier to use and result in cleaner build files. As an added benefit, our use of `getProject().files()` means that our custom method can set up an inferred task dependency.

One last thing to note: if you are developing a task that takes collections of source files as inputs, like this example, consider using the built-in `SourceTask`. It will save you having to implement some of the plumbing that we put into `ProcessTemplates`.

Linking an @OutputDirectory to an @InputFiles

When you want to link the output of one task to the input of another, the types often match and a simple property assignment will provide that link. For example, a `File` output property can be assigned to a `File` input.

Unfortunately, this approach breaks down when you want the files in a task's `@OutputDirectory` (of type `File`) to become the source for another task's `@InputFiles` property (of type `FileCollection`). Since the two have different types, property assignment won't work.

As an example, imagine you want to use the output of a Java compilation task - via the `destinationDir` property - as the input of a custom task that instruments a set of files containing Java bytecode. This custom task, which we'll call `Instrument`, has a `classFiles` property annotated with `@InputFiles`. You might initially try to configure the task like so:

Example 19.31. Failed attempt at setting up an inferred task dependency

build.gradle

```
apply plugin: "java"

task badInstrumentClasses(type: Instrument) {
    classFiles = fileTree(compileJava.destinationDir)
    destinationDir = file("$buildDir/instrumented")
}
```

Output of **gradle clean badInstrumentClasses**

```
> gradle clean badInstrumentClasses
:clean UP-TO-DATE
:badInstrumentClasses NO-SOURCE

BUILD SUCCESSFUL
```

There's nothing obviously wrong with this code, but you can see from the console output that the compilation task is missing. In this case you would need to add an explicit task dependency between `instrumentClasses` and `compileJava` via `dependsOn`. The use of `fileTree()` means that Gradle can't infer the task dependency itself.

One solution is to use the `TaskOutputs.files` property, as demonstrated by the following example:

Example 19.32. Setting up an inferred task dependency between output dir and input files

build.gradle

```
task instrumentClasses(type: Instrument) {
    classFiles = compileJava.outputs.files
    destinationDir = file("$buildDir/instrumented")
}
```

Output of **gradle clean instrumentClasses**

```
> gradle clean instrumentClasses
:clean UP-TO-DATE
:compileJava
:instrumentClasses

BUILD SUCCESSFUL
```

Alternatively, you can get Gradle to access the appropriate property itself by using the `project.files()` method in place of `project.fileTree()`:

Example 19.33. Setting up an inferred task dependency with files()

build.gradle

```
task instrumentClasses2(type: Instrument) {
    classFiles = files(compileJava)
    destinationDir = file("$buildDir/instrumented")
}
```

Output of **gradle clean instrumentClasses2**

```
> gradle clean instrumentClasses2
:clean UP-TO-DATE
:compileJava
:instrumentClasses2

BUILD SUCCESSFUL
```

Remember that `files()` can take tasks as arguments, whereas `fileTree()` cannot.

The downside of this approach is that all file outputs of the source task become the input files of the target - `instrumentClasses2` in this case. That's fine as long as the source task only has a single file-based output, like the `JavaCompile` task. But if you have to link just one output property among several, then you need to explicitly tell Gradle which task generates the input files using the `builtBy` method:

Example 19.34. Setting up an inferred task dependency with builtBy()

build.gradle

```
task instrumentClassesBuiltBy(type: Instrument) {
    classFiles = fileTree(compileJava.destinationDir) {
        builtBy compileJava
    }
    destinationDir = file("$buildDir/instrumented")
}
```

Output of **gradle clean instrumentClassesBuiltBy**

```
> gradle clean instrumentClassesBuiltBy
:clean UP-TO-DATE
:compileJava
:instrumentClassesBuiltBy

BUILD SUCCESSFUL
```

You can of course just add an explicit task dependency via `dependsOn`, but the above approach provides more semantic meaning, explaining why `compileJava` has to run beforehand.

Providing custom up-to-date logic

Gradle automatically handles up-to-date checks for output files and directories, but what if the task output is something else entirely? Perhaps it's an update to a web service or a database table. Gradle has no way of knowing how to check whether the task is up to date in such cases.

That's where the `upToDateWhen()` method on `TaskOutputs` comes in. This takes a predicate function

that is used to determine whether a task is up to date or not. One use case is to disable up-to-date checks completely for a task, like so:

Example 19.35. Ignoring up-to-date checks

build.gradle

```
task alwaysInstrumentClasses(type: Instrument) {
    classFiles = files(compileJava)
    destinationDir = file("$buildDir/instrumented")
    outputs.upToDateWhen { false }
}
```

Output of **gradle clean alwaysInstrumentClasses**

```
> gradle clean alwaysInstrumentClasses
:compileJava
:alwaysInstrumentClasses
```

BUILD SUCCESSFUL

Output of **gradle alwaysInstrumentClasses**

```
> gradle alwaysInstrumentClasses
:compileJava UP-TO-DATE
:alwaysInstrumentClasses
```

BUILD SUCCESSFUL

The `{ false }` closure ensures that `copyResources` will always perform the copy, irrespective of whether there is no change in the inputs or outputs.

You can of course put more complex logic into the closure. You could check whether a particular record in a database table exists or has changed for example. Just be aware that up-to-date checks should *_save_* you time. Don't add checks that cost as much or more time than the standard execution of the task. In fact, if a task ends up running frequently anyway, because it's rarely up to date, then it may not be worth having an up-to-date check at all. Remember that your checks will always run if the task is in the execution task graph.

One common mistake is to use `upToDateWhen()` instead of `Task.onlyIf()`. If you want to skip a task on the basis of some condition unrelated to the task inputs and outputs, then you should use `onlyIf()`. For example, in cases where you want to skip a task when a particular property is set or not set.

19.9.3. How does it work?

Before a task is executed for the first time, Gradle takes a snapshot of the inputs. This snapshot contains the paths of input files and a hash of the contents of each file. Gradle then executes the task. If the task completes successfully, Gradle takes a snapshot of the outputs. This snapshot contains the set of output files and a hash of the contents of each file. Gradle persists both snapshots for the next time the task is executed.

Each time after that, before the task is executed, Gradle takes a new snapshot of the inputs and outputs. If the new snapshots are the same as the previous snapshots, Gradle assumes that the outputs are up to date and skips the task. If they are not the same, Gradle executes the task. Gradle persists both snapshots for the next time the task is executed.

Gradle also considers the *code* of the task as part of the inputs to the task. When a task, its actions, or its dependencies change between executions, Gradle considers the task as out-of-date.

Gradle understands if a file property (e.g. one holding a Java classpath) is order-sensitive. When comparing the snapshot of such a property, even a change in the order of the files will result in the task becoming out-of-date.

Note that if a task has an output directory specified, any files added to that directory since the last time it was executed are ignored and will NOT cause the task to be out of date. This is so unrelated tasks may share an output directory without interfering with each other. If this is not the behaviour you want for some reason, consider using `TaskOutputs.upToDateWhen(groovy.lang.Closure)`

19.10. Task rules

Sometimes you want to have a task whose 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 19.36. Task rule

build.gradle

```
tasks.addRule("Pattern: ping<ID>") { String taskName ->
    if (taskName.startsWith("ping")) {
        task(taskName) {
            doLast {
                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, which is shown with **gradle tasks**.

Rules are not only used when calling tasks from the command line. You can also create `dependsOn` relations on rule based tasks:

Example 19.37. Dependency on rule based tasks

build.gradle

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

task groupPing {
    dependsOn pingServer1, pingServer2
}
```

Output of `gradle -q groupPing`

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

If you run “`gradle -q tasks`” you won't find a task named “`pingServer1`” or “`pingServer2`”, but this script is executing logic based on the request to run those tasks.

19.11. Finalizer tasks

Finalizers tasks are an *incubating* feature (see Section C.1.2, “Incubating”).

Finalizer tasks are automatically added to the task graph when the finalized task is scheduled to run.

Example 19.38. Adding a task finalizer

build.gradle

```
task taskX {
    doLast {
        println 'taskX'
    }
}
task taskY {
    doLast {
        println 'taskY'
    }
}

taskX.finalizedBy taskY
```

Output of **gradle -q taskX**

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

Finalizer tasks will be executed even if the finalized task fails.

Example 19.39. Task finalizer for a failing task

build.gradle

```
task taskX {
    doLast {
        println 'taskX'
        throw new RuntimeException()
    }
}
task taskY {
    doLast {
        println 'taskY'
    }
}

taskX.finalizedBy taskY
```

Output of **gradle -q taskX**

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

On the other hand, finalizer tasks are not executed if the finalized task didn't do any work, for example if it is considered up to date or if a dependent task fails.

Finalizer tasks are useful in situations where the build creates a resource that has to be cleaned up regardless of the build failing or succeeding. An example of such a resource is a web container that is started before an integration test task and which should be always shut down, even if some of the tests fail.

To specify a finalizer task you use the `Task.finalizedBy(java.lang.Object[])` method. This

method accepts a task instance, a task name, or any other input accepted by `Task.dependsOn(java.lang.Object[])`.

19.12. Summary

If you are coming from Ant, an enhanced Gradle task like *Copy* seems like a cross between an Ant target and an Ant task. Although Ant's tasks and targets are really different entities, Gradle combines these notions into a single entity. Simple Gradle tasks are like Ant's targets, but enhanced Gradle tasks also include aspects of Ant tasks. All of Gradle's tasks share a common API and you can create dependencies between them. These tasks are much easier to configure than an Ant task. They make full use of the type system, and are more expressive and easier to maintain.

[7] 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 (see Section 18.8, “Default imports”).

[8] You will also see `UP-TO-DATE` next to tasks that have no actions, even though that's nothing to do with incremental build.

20

Working With Files

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

20.1. Locating files

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

Example 20.1. 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, and it will attempt to convert the value to an absolute `File` object. Usually, you would pass it a `String` or `File` instance. 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. The `file()` method also understands URLs, such as `file:/some/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 can change depending on how the user runs Gradle.

20.2. File collections

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

One way to obtain a `FileCollection` instance is to use the

`Project.files(java.lang.Object[])` method. You can pass this method any number of objects, which are then converted into a set of `File` objects. The `files()` method accepts any type of object as its parameters. These are evaluated relative to the project directory, as per the `file()` method, described in Section 20.1, “Locating files”. You can also pass collections, iterables, maps and arrays to the `files()` method. These are flattened and the contents converted to `File` instances.

Example 20.2. Creating a file collection

build.gradle

```
FileCollection collection = files('src/file1.txt',
                                new File('src/file2.txt'),
                                ['src/file3.txt', 'src/file4.txt'])
```

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, or subtract one file collection from another using the `-` operator. Here are some examples of what you can do with a file collection.

Example 20.3. Using a file collection

build.gradle

```
// Iterate over the files in the collection
collection.each { File file ->
    println 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 and subtract collections
def union = collection + files('src/file3.txt')
def different = 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 20.4. Implementing a file collection

build.gradle

```
task list {
    doLast {
        File srcDir

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

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

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

Output of `gradle -q list`

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

Some other types of things you can pass to `files()`:

FileCollection

These are flattened and the contents included in the file collection.

Task

The output files of the task are included in the file collection.

TaskOutputs

The output files of the TaskOutputs are included in the file collection.

It is important to note that the content of a file collection is evaluated lazily, when it is needed. This means you can, for example, create a `FileCollection` that represents files which will be created in the future by, say, some task.

20.3. File trees

A *file tree* is a collection of files arranged in a hierarchy. For example, a file tree might represent a directory tree or the contents of a ZIP file. 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(java.util.Map)` method. This creates a `FileTree` defined with a base directory, and optionally some Ant-style include and exclude patterns.

Example 20.5. Creating a file tree

build.gradle

```
// Create a file tree with a base directory
FileTree 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('src') {
    include '**/*.java'
}

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

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 sub-tree using Ant-style patterns:

Example 20.6. Using a file tree

build.gradle

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

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

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

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

20.4. Using the contents of an archive as a file tree

You can use the contents of an archive, such as a ZIP or TAR file, as a file tree. You do this using the `Project.zipTree(java.lang.Object)` and `Project.tarTree(java.lang.Object)` methods. These methods return a `FileTree` instance which you can use like any other file tree or file collection. For example, you can use it to expand the archive by copying the contents, or to merge some archives into another.

Example 20.7. Using an archive as a file tree

build.gradle

```
// Create a ZIP file tree using path
FileTree zip = zipTree('someFile.zip')

// Create a TAR file tree using path
FileTree tar = tarTree('someFile.tar')

//tar tree attempts to guess the compression based on the file extension
//however if you must specify the compression explicitly you can:
FileTree someTar = tarTree(resources.gzip('someTar.ext'))
```

20.5. Specifying a set of input files

Many objects in Gradle have properties which accept a set of input files. For example, the `JavaCompile` 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 was shown above. This means you can set the property using, for example, a `File`, `String`, collection, `FileCollection` or even a closure. Here are some examples:

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 20.8. Specifying a set of files

build.gradle

```
task compile(type: JavaCompile)

// Use a File object to specify the source directory
compile {
    source = file('src/main/java')
}

// Use a String path to specify the source directory
compile {
    source = 'src/main/java'
}

// Use a collection to specify multiple source directories
compile {
    source = ['src/main/java', '../shared/java']
}

// Use a FileCollection (or FileTree in this case) to specify the source files
compile {
    source = fileTree(dir: 'src/main/java').matching { include 'org/gradle/api/**'
}

// Using a closure to specify the source files.
compile {
    source = {
        // Use the contents of each zip file in the src dir
        file('src').listFiles().findAll { it.name.endsWith('.zip') }.collect { zip
    }
}
```

build.gradle

```
compile {
    // Add some source directories use String paths
    source 'src/main/java', 'src/main/groovy'

    // Add a source directory using a File object
    source file('../shared/java')

    // Add some source directories using a closure
    source { file('src/test/').listFiles() }
}
```

20.6. Copying files

You can use the Copy task to copy files. The copy task is very flexible, and allows you to, for example, filter the contents of the files as they are copied, and map to the file names.

To use the Copy task, you must provide a set of source files to copy, and a destination directory to copy the files to. You may also specify how to transform the files as they are copied. You do all this using a *copy spec*. A copy spec is represented by the CopySpec interface. The Copy task implements this interface.

You specify the source files using the `CopySpec.from(java.lang.Object[])` method. To specify the destination directory, use the `CopySpec.into(java.lang.Object)` method.

Example 20.9. Copying files using the copy task

build.gradle

```
task copyTask(type: Copy) {
    from 'src/main/webapp'
    into 'build/explodedWar'
}
```

The `from()` method accepts any of the arguments that the `files()` method does. When an argument resolves to a directory, everything under that directory (but not the directory itself) is recursively copied into the destination directory. When an argument resolves to a file, that file is copied into the destination directory. When an argument resolves to a non-existing file, that argument is ignored. If the argument is a task, the output files (i.e. the files the task creates) of the task are copied and the task is automatically added as a dependency of the Copy task. The `into()` accepts any of the arguments that the `file()` method does. Here is another example:

Example 20.10. Specifying copy task source files and destination directory

build.gradle

```
task anotherCopyTask(type: Copy) {
    // Copy everything under src/main/webapp
    from 'src/main/webapp'
    // Copy a single file
    from 'src/staging/index.html'
    // Copy the output of a task
    from copyTask
    // Copy the output of a task using Task outputs explicitly.
    from copyTaskWithPatterns.outputs
    // Copy the contents of a Zip file
    from zipTree('src/main/assets.zip')
    // Determine the destination directory later
    into { getDestDir() }
}
```

You can select the files to copy using Ant-style include or exclude patterns, or using a closure:

Example 20.11. Selecting the files to copy

build.gradle

```
task copyTaskWithPatterns(type: Copy) {
    from 'src/main/webapp'
    into 'build/explodedWar'
    include '**/*.html'
    include '**/*.jsp'
    exclude { details -> details.file.name.endsWith('.html') &&
        details.file.text.contains('staging') }
}
```

You can also use the `Project.copy(org.gradle.api.Action)` method to copy files. It works the

same way as the task with some major limitations though. First, the `copy()` is not incremental (see Section 19.9, “Up-to-date checks (AKA Incremental Build)”).

Example 20.12. Copying files using the `copy()` method without up-to-date check

build.gradle

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

Secondly, the `copy()` method can not honor task dependencies when a task is used as a copy source (i.e. as an argument to `from()`) because it's a method and not a task. As such, if you are using the `copy()` method as part of a task action, you must explicitly declare all inputs and outputs in order to get the correct behavior.

Example 20.13. Copying files using the `copy()` method with up-to-date check

build.gradle

```
task copyMethodWithExplicitDependencies{
    // up-to-date check for inputs, plus add copyTask as dependency
    inputs.file copyTask
    outputs.dir 'some-dir' // up-to-date check for outputs
    doLast{
        copy {
            // Copy the output of copyTask
            from copyTask
            into 'some-dir'
        }
    }
}
```

It is preferable to use the `Copy` task wherever possible, as it supports incremental building and task dependency inference without any extra effort on your part. The `copy()` method can be used to copy files as *part* of a task's implementation. That is, the `copy` method is intended to be used by custom tasks (see Chapter 40, *Writing Custom Task Classes*) that need to copy files as part of their function. In such a scenario, the custom task should sufficiently declare the inputs/outputs relevant to the copy action.

20.6.1. Renaming files

Example 20.14. Renaming files as they are copied

build.gradle

```
task rename(type: Copy) {
    from 'src/main/webapp'
    into 'build/explodedWar'
    // Use a closure to map the file name
    rename { String fileName ->
        fileName.replace('-staging-', '')
    }
    // Use a regular expression to map the file name
    rename '(.+)-staging-(.+)', '$1$2'
    rename(/(.+)-staging-(.+)/, '$1$2')
}
```

20.6.2. Filtering files

Example 20.15. Filtering files as they are copied

build.gradle

```
import org.apache.tools.ant.filters.FixCrLfFilter
import org.apache.tools.ant.filters.ReplaceTokens

task filter(type: Copy) {
    from 'src/main/webapp'
    into 'build/explodedWar'
    // Substitute property tokens in files
    expand(copyright: '2009', version: '2.3.1')
    expand(project.properties)
    // Use some of the filters provided by Ant
    filter(FixCrLfFilter)
    filter(ReplaceTokens, tokens: [copyright: '2009', version: '2.3.1'])
    // Use a closure to filter each line
    filter { String line ->
        "[$line]"
    }
    // Use a closure to remove lines
    filter { String line ->
        line.startsWith('-') ? null : line
    }
    filteringCharset = 'UTF-8'
}
```

When you use the `ReplaceTokens` class with the “filter” operation, the result is a template engine that replaces tokens of the form “@tokenName@” (the Apache Ant-style token) with a set of given values. The “expand” operation does the same thing except it treats the source files as Groovy templates in which tokens take the form “\${tokenName}”. Be aware that you may need to escape parts of your source files when using this option, for example if it contains literal “\$” or “<%” strings.

It's a good practice to specify the charset when reading and writing the file, using the `filteringCharset`

property. If not specified, the JVM default charset is used, which might not match with the actual charset of the files to filter, and might be different from one machine to another.

20.6.3. Using the CopySpec class

Copy specs form a hierarchy. A copy spec inherits its destination path, include patterns, exclude patterns, copy actions, name mappings and filters.

Example 20.16. Nested copy specs

build.gradle

```
task nestedSpecs(type: Copy) {
    into 'build/explodedWar'
    exclude '**/*staging*'
    from('src/dist') {
        include '**/*.html'
    }
    into('libs') {
        from configurations.runtime
    }
}
```

20.7. Using the Sync task

The Sync task extends the Copy task. When it executes, it copies the source files into the destination directory, and then removes any files from the destination directory which it did not copy. This can be useful for doing things such as installing your application, creating an exploded copy of your archives, or maintaining a copy of the project's dependencies.

Here is an example which maintains a copy of the project's runtime dependencies in the build/libs directory.

Example 20.17. Using the Sync task to copy dependencies

build.gradle

```
task libs(type: Sync) {
    from configurations.runtime
    into "$buildDir/libs"
}
```

20.8. Creating archives

A project can have as many JAR archives as you want. You can also add WAR, ZIP and TAR archives to your project. Archives are created using the various archive tasks: Zip, Tar, Jar, War, and Ear. They all work the same way, so let's look at how you create a ZIP file.

Example 20.18. Creating a ZIP archive

build.gradle

```
apply plugin: 'java'

task zip(type: Zip) {
    from 'src/dist'
    into('libs') {
        from configurations.runtime
    }
}
```

The archive tasks all work exactly the same way as the Copy task, and implement the same CopySpec interface. As with the Copy task, you specify the input files using the `from()` method, and can optionally specify where they end up in the archive using the `into()` method. You can filter the contents of file, rename files, and all the other things you can do with a copy spec.

20.8.1. Archive naming

The format of *projectName-version.type* is used for generated archive file names. For example:

Example 20.19. Creation of ZIP archive

build.gradle

```
apply plugin: 'java'

version = 1.0

task myZip(type: Zip) {
    from 'somedir'
}

println myZip.archiveName
println relativePath(myZip.destinationDir)
println relativePath(myZip.archivePath)
```

Output of **gradle -q myZip**

```
> gradle -q myZip
zipProject-1.0.zip
build/distributions
build/distributions/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 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.

Why are you using the Java plugin?

The Java plugin adds a number of default values for the archive tasks. You can use the archive tasks without using the Java plugin, if you like. You will need to provide values for some additional properties.

There are a number of properties which you can set on an archive task. These are listed below in Table 20.1, “Archive tasks - naming properties”. You can, for example, change the name of the archive:

Example 20.20. Configuration of archive task - custom archive name

build.gradle

```
apply plugin: 'java'
version = 1.0

task myZip(type: Zip) {
    from '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 20.21. Configuration of archive task - appendix & classifier

build.gradle

```
apply plugin: 'java'
archivesBaseName = 'gradle'
version = 1.0

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

println myZip.archiveName
```

Output of **gradle -q myZip**

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

Table 20.1. Archive tasks - naming properties

Property name	Type	Default value	Description
archiveName	String	<i>baseName-appendix-version-classifier</i> If any of these properties is empty the trailing - is not added to the name.	The base file name of the generated archive
archivePath	File	<i>destinationDir/archiveName</i>	The absolute path of the generated archive.
destinationDir	File	Depends on the archive type. JARs and WARs go into <i>project.buildDir/libraries</i> . ZIPs and TARs go into <i>project.buildDir/distributions</i> .	The directory to generate the archive into
baseName	String	<i>project.name</i>	The base name portion of the archive file name.
appendix	String	null	The appendix portion of the archive file name.
version	String	<i>project.version</i>	The version portion of the archive file name.
classifier	String	null	The classifier portion of the archive file name,
extension	String	Depends on the archive type, and for TAR files, the compression type as well: zip, jar, war, tar, tgz or tbz2.	The extension of the archive file name.

20.8.2. Sharing content between multiple archives

You can use the `Project.copySpec(org.gradle.api.Action)` method to share content between archives.

20.8.3. Reproducible archives

Sometimes it can be desirable to recreate archives in a byte for byte way on different machines. You want to be sure that building an artifact from source code produces the same result, byte for byte, no matter when and where it is built. This is necessary for projects like reproducible-builds.org.

Reproducing the same archive byte for byte poses some challenges since the order of the files in an archive is influenced by the underlying filesystem. Each time a zip, tar, jar, war or ear is built from source, the order of the files inside the archive may change. Files that only have a different timestamp also causes archives to be slightly different between builds. All `AbstractArchiveTask` (e.g. Jar, Zip) tasks shipped with Gradle include incubating support producing reproducible archives.

For example, to make a Zip task reproducible you need to set `Zip.isReproducibleFileOrder()` to `true` and `Zip.isPreserveFileTimestamps()` to `false`. In order to make all archive tasks in your build reproducible, consider adding the following configuration to your build file:

Example 20.22. Activating reproducible archives

build.gradle

```
tasks.withType(AbstractArchiveTask) {
    preserveFileTimestamps = false
    reproducibleFileOrder = true
}
```

Often you will want to publish an archive, so that it is usable from another project. This process is described in Chapter 32, *Publishing artifacts*

20.9. Properties files

Properties files are used in many places during Java development. Gradle makes it easy to create properties files as a normal part of the build. You can use the `WriteProperties` task to create properties files.

The `WriteProperties` task also fixes a well-known problem with `Properties.store()` that can reduce the usefulness of incremental builds (see Section 19.9, “Up-to-date checks (AKA Incremental Build)”). The standard Java way to write a properties file produces a unique file every time, even when the same properties and values are used, because it includes a timestamp in the comments. Gradle's `WriteProperties` task generates exactly the same output byte-for-byte if none of the properties have changed. This is achieved by a few tweaks to how a properties file is generated:

- no timestamp comment is added to the output
- the line separator is system independent, but can be configured explicitly (it defaults to `'\n'`)

- the properties are sorted alphabetically

21

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` file, 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()`.^[9]

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.

21.1. 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 of the `echo` task. Notice that we can also mix Groovy code and the Ant task markup. This can be extremely powerful.

Example 21.1. Using an Ant task

build.gradle

```
task hello {
    doLast {
        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 21.2. Passing nested text to an Ant task

build.gradle

```
task hello {
    doLast {
        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 21.3. Passing nested elements to an Ant task

build.gradle

```
task zip {
    doLast {
        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 21.4. Using an Ant type

build.gradle

```
task list {
    doLast {
        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](#)

21.1.1. Using custom Ant tasks in your build

To make custom tasks available in your build, you can use the `taskdef` (usually easier) or `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 21.5. Using a custom Ant task

build.gradle

```
task check {
    doLast {
        ant.taskdef(resource: 'checkstyletask.properties') {
            classpath {
                fileset(dir: 'libs', includes: '*.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 25.4, “How to declare your dependencies”.

Example 21.6. Declaring the classpath for a custom Ant task

build.gradle

```
configurations {
    pmd
}

dependencies {
    pmd group: 'pmd', name: 'pmd', version: '4.2.5'
}
```

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

Example 21.7. Using a custom Ant task and dependency management together

build.gradle

```
task check {
    doLast {
        ant.taskdef(name: 'pmd',
                    classname: 'net.sourceforge.pmd.ant.PMDTask',
                    classpath: configurations.pmd.asPath)
        ant.pmd(shortFileNames: 'true',
                failonruleviolation: 'true',
                rulesetfiles: file('pmd-rules.xml').toURI().toString()) {
            formatter(type: 'text', toConsole: 'true')
            fileset(dir: 'src')
        }
    }
}
```

21.2. 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 21.8. 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 21.9. Task that depends on Ant target

build.gradle

```
ant.importBuild 'build.xml'

task intro(dependsOn: hello){
  doLast {
    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 21.10. Adding behaviour to an Ant target

build.gradle

```
ant.importBuild 'build.xml'

hello {
    doLast {
        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 21.11. Ant target that depends on Gradle task

build.gradle

```
ant.importBuild 'build.xml'

task intro {
    doLast {
        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
```


Sometimes it may be necessary to “rename” the task generated for an Ant target to avoid a naming collision with existing Gradle tasks. To do this, use the `AntBuilder.importBuild(java.lang.Object, org.gradle.api.Transformer)` method.

Example 21.12. Renaming imported Ant targets

build.gradle

```
ant.importBuild('build.xml') { antTargetName ->
    'a-' + antTargetName
}
```

build.xml

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

Output of `gradle a-hello`

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

BUILD SUCCESSFUL

Total time: 1 secs
```

Note that while the second argument to this method should be a `Transformer`, when programming in Groovy we can simply use a closure instead of an anonymous inner class (or similar) due to Groovy's support for automatically coercing closures to single-abstract-method types.

21.3. 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 21.13. 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 21.14. 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 21.15. 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 21.16. Getting an Ant reference

build.xml

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

build.gradle

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

21.4. Ant logging

Gradle maps Ant message priorities to Gradle log levels so that messages logged from Ant appear in the Gradle output. By default, these are mapped as follows:

Table 21.1. Ant message priority mapping

Ant Message Priority	Gradle Log Level
<i>VERBOSE</i>	DEBUG
<i>DEBUG</i>	DEBUG
<i>INFO</i>	INFO
<i>WARN</i>	WARN
<i>ERROR</i>	ERROR

21.4.1. Fine tuning Ant logging

The default mapping of Ant message priority to Gradle log level can sometimes be problematic. For example, there is no message priority that maps directly to the `LIFECYCLE` log level, which is the default for Gradle. Many Ant tasks log messages at the *INFO* priority, which means to expose those messages from Gradle, a build would have to be run with the log level set to `INFO`, potentially logging much more output than is desired.

Conversely, if an Ant task logs messages at too high of a level, to suppress those messages would require the build to be run at a higher log level, such as `QUIET`. However, this could result in other, desirable output being suppressed.

To help with this, Gradle allows the user to fine tune the Ant logging and control the mapping of message priority to Gradle log level. This is done by setting the priority that should map to the default Gradle `LIFECYCLE` log level using the `AntBuilder.setLifecycleLogLevel(java.lang.String)` method. When this value is set, any Ant message logged at the configured priority or above will be logged at least at `LIFECYCLE`. Any Ant message logged below this priority will be logged at most at `INFO`.

For example, the following changes the mapping such that Ant *INFO* priority messages are exposed at the `LIFEC` log level.

Example 21.17. Fine tuning Ant logging

build.gradle

```
ant.lifecycleLogLevel = "INFO"

task hello {
    doLast {
        ant.echo(level: "info", message: "hello from info priority!")
    }
}
```

Output of **gradle hello**

```
> gradle hello
:hello
[ant:echo] hello from info priority!

BUILD SUCCESSFUL

Total time: 1 secs
```

On the other hand, if the `lifecycleLogLevel` was set to *ERROR*, Ant messages logged at the *WARN* priority would no longer be logged at the WARN log level. They would now be logged at the INFO level and would be suppressed by default.

21.5. API

The Ant integration is provided by `AntBuilder`.

[9] 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

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. These 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*.

22.1. 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

During this phase the project objects are configured. The build scripts of *all* projects which are part of the build are executed. Gradle 1.4 introduced an incubating opt-in feature called *configuration on demand*. In this mode, Gradle configures only relevant projects (see the section called “Configuration on demand”).

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.

22.2. 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 is executed during the initialization phase. A multiproject build must have a `settings.gradle` file in the root project of the multiproject hierarchy. It is required because the settings file defines which

projects are taking part in the multi-project build (see Chapter 26, *Multi-project Builds*). For a single-project build, a settings file is optional. Besides defining the included projects, you might need it to add libraries to your build script classpath (see Chapter 43, *Organizing Build Logic*). Let's first do some introspection with a single project build:

Example 22.1. 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 {
    doLast {
        println 'This is executed during the execution phase.'
    }
}

task testBoth {
    doFirst {
        println 'This is executed first during the execution phase.'
    }
    doLast {
        println 'This is executed last during the execution phase.'
    }
    println 'This is executed during the configuration phase as well.'
}
```

Output of **gradle test testBoth**

```
> gradle test testBoth
This is executed during the initialization phase.
This is executed during the configuration phase.
This is also executed during the configuration phase.
This is executed during the configuration phase as well.
:test
This is executed during the execution phase.
:testBoth
This is executed first during the execution phase.
This is executed last 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. Look at the `Settings` class in the API documentation for more information.

22.3. 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 26, *Multi-project Builds*).

22.3.1. Project locations

Multi-project builds are always represented by a tree with a single root. Each element in the tree represents a project. A project has a path which denotes the position of the project in the multi-project build tree. In most cases the project path is consistent with the physical location of the project in the file system. However, this behavior is configurable. 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.

22.3.2. 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.

Hierarchical layouts

Example 22.2. Hierarchical layout

settings.gradle

```
include 'project1', 'project2:child', 'project3:child1'
```

The `include` method takes project paths as arguments. The project path is assumed to be equal to the relative physical file system path. For example, a path `'services:api'` is mapped by default to a folder `'services/api'` (relative from the project root). You only need to specify the leaves of the tree. This means that the inclusion of the path `'services:hotels:api'` will result in creating 3 projects: `'services'`, `'services:hotels'` and `'services:hotels:api'`.

Flat layouts

Example 22.3. Flat layout

settings.gradle

```
includeFlat 'project3', 'project4'
```

The `includeFlat` method takes directory names as an argument. These directories need to exist as siblings of the root project directory. The location of these directories are considered as child projects of the root project in the multi-project tree.

22.3.3. 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:

Using this descriptor you can change the name, project directory and build file of a project.

Example 22.4. Modification of elements of the project tree

settings.gradle

```
println rootProject.name
println project(':projectA').name
```

settings.gradle

```
rootProject.name = 'main'
project(':projectA').projectDir = new File(settingsDir, '../my-project-a')
project(':projectA').buildFileName = 'projectA.gradle'
```

Look at the `ProjectDescriptor` class in the API documentation for more information.

22.4. Initialization

How does Gradle know whether to do a single or multiproject build? If you trigger a multiproject build from a directory with a settings file, things are easy. But Gradle also allows you to execute the build from within any subproject taking part in the build.^[10] If you execute Gradle from within a project with no `settings.gradle` file, Gradle looks for a `settings.gradle` file in the following way:

- It looks in a directory called `master` which has the same nesting level as the current dir.
- If not found yet, it searches parent directories.
- If not found yet, 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? Gradle needs to determine whether the project you are in is a subproject of a multiproject build or not. Of course, if it is a subproject, only the subproject and its dependent projects are built, but Gradle needs to create the build configuration for the whole multiproject build (see Chapter 26, *Multi-project Builds*). You can use the `-u` command line option to tell Gradle not to look in the parent hierarchy for a `settings.gradle` file. The current project is then always built 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 automatic search for a `settings.gradle` file only works for multi-project builds with a physical hierarchical or flat layout. For a flat layout you must additionally follow the naming convention described

above (“master”). 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 information on how to run partial builds from the root see Section 26.4, “Running tasks by their absolute path”.

Gradle creates a Project object for every project taking part in the build. For a multi-project build these are the projects specified in the Settings object (plus the root project). Each project object has by default a name equal to the name of its top level directory, and every project except the root project has a parent project. Any project may have child projects.

22.5. 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 26, *Multi-project Builds*.

22.6. 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 two 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.

22.6.1. 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 which has a `hasTests` property value of `true`.

Example 22.5. 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') {
                doLast {
                    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 22.6. Notifications

build.gradle

```
gradle.afterProject {project, projectState ->
    if (projectState.failure) {
        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.

22.6.2. 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 22.7. Setting of certain property to all tasks

build.gradle

```
tasks.whenTaskAdded { task ->
    task.ext.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 an `Action` to a `TaskContainer` to receive these events.

22.6.3. 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 16.13, “Configure by DAG”.

You can also add a `TaskExecutionGraphListener` to the `TaskExecutionGraph` to receive these events.

22.6.4. 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 22.8. Logging of start and end of each task execution

build.gradle

```
task ok

task broken(dependsOn: ok) {
    doLast {
        throw new RuntimeException('broken')
    }
}

gradle.taskGraph.beforeTask { Task task ->
    println "executing $task ..."
}

gradle.taskGraph.afterTask { Task task, TaskState state ->
    if (state.failure) {
        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.

[10] Gradle supports partial multiproject builds (see Chapter 26, *Multi-project Builds*).

23

Wrapper Plugin

The wrapper plugin is currently incubating. Please be aware that the DSL and other configuration may change in later Gradle versions.

The Gradle wrapper plugin allows the generation of Gradle wrapper files by adding a `Wrapper` task, that generates all files needed to run the build using the Gradle Wrapper. Details about the Gradle Wrapper can be found in Chapter 5, *The Gradle Wrapper*.

23.1. Usage

Without modifying the `build.gradle` file, the wrapper plugin can be auto-applied to the root project of the current build by running “`gradle wrapper`” from the command line. This applies the plugin if no task named `wrapper` is already defined in the build.

23.2. Tasks

The wrapper plugin adds the following tasks to the project:

Table 23.1. Wrapper plugin - tasks

Task name	Depends on	Type	Description
wrapper	-	Wrapper	Generates Gradle wrapper files.

24

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 relevant information for figuring out if things have gone wrong. Gradle defines 6 log levels, as shown in Table 24.1, “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 24.1. 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

24.1. Choosing a log level

You can use the command line switches shown in Table 24.2, “Log level command-line options” to choose different log levels. In Table 24.3, “Stacktrace command-line options” you find the command line switches which affect stacktrace logging.

Table 24.2. Log level command-line options

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

Table 24.3. 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 <code>DEBUG</code> log level is chosen, truncated stacktraces are always printed.
<code>-s</code> or <code>--stacktrace</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.) This option renders stacktraces for deprecation warnings.
<code>-S</code> or <code>--full-stacktrace</code>	The full stacktraces are printed out. This option renders stacktraces for deprecation warnings.

24.2. 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 its logging system at the `QUIET` log level.

Example 24.1. 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 24.2. 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 `buildSrc` 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 24.3. Using SLF4J to write log messages

build.gradle

```
import org.slf4j.Logger
import org.slf4j.LoggerFactory

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

24.3. 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. The project object provides a `LoggingManager`, which allows you to change the log levels that standard out or error are redirected to when your build script is evaluated.

Example 24.4. Configuring standard output capture

build.gradle

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

To change the log level for standard out or error during task execution, tasks also provide a `LoggingManager`.

Example 24.5. Configuring standard output capture for a task

build.gradle

```
task logInfo {
    logging.captureStandardOutput LogLevel.INFO
    doFirst {
        println 'A task message which is logged at INFO level'
    }
}
```

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.

24.4. Changing what Gradle logs

You can replace much of Gradle's logging UI with your own. You might do this, for example, if you want to customize the UI in some way - to log more or less information, or to change the formatting. You replace the logging using the `Gradle.useLogger(java.lang.Object)` method. This is accessible from a build script, or an init script, or via the embedding API. Note that this completely disables Gradle's default output. Below is an example init script which changes how task execution and build completion is logged.

Example 24.6. Customizing what Gradle logs

init.gradle

```
useLogger(new CustomEventLogger())

class CustomEventLogger extends BuildAdapter implements TaskExecutionListener {

    public void beforeExecute(Task task) {
        println "[$task.name]"
    }

    public void afterExecute(Task task, TaskState state) {
        println()
    }

    public void buildFinished(BuildResult result) {
        println 'build completed'
        if (result.failure != null) {
            result.failure.printStackTrace()
        }
    }
}
```

Output of `gradle -I init.gradle build`

```
> gradle -I init.gradle build
[compile]
compiling source

[testCompile]
compiling test source

[test]
running unit tests

[build]

build completed
```

Your logger can implement any of the listener interfaces listed below. When you register a logger, only the logging for the interfaces that it implements is replaced. Logging for the other interfaces is left untouched. You can find out more about the listener interfaces in Section 22.6, “Responding to the lifecycle in the build script”.

- BuildListener

- ProjectEvaluationListener
- TaskExecutionGraphListener
- TaskExecutionListener
- TaskActionListener

Dependency Management

25.1. Introduction

Dependency management is a critical feature of every build, and Gradle has placed an emphasis on offering first-class dependency management that is both easy to understand and compatible with a wide variety of approaches. If you are familiar with the approach used by either Maven or Ivy you will be delighted to learn that Gradle is fully compatible with both approaches in addition to being flexible enough to support fully-customized approaches.

Here are the major highlights of Gradle's support for dependency management:

- Transitive dependency management: Gradle gives you full control of your project's dependency tree.
- Support for non-managed dependencies: If your dependencies are simply files in version control or a shared drive, Gradle provides powerful functionality to support this.
- Support for custom dependency definitions: Gradle's Module Dependencies give you the ability to describe the dependency hierarchy in the build script.
- A fully customizable approach to Dependency Resolution: Gradle provides you with the ability to customize resolution rules making dependency substitution easy.
- Full Compatibility with Maven and Ivy: If you have defined dependencies in a Maven POM or an Ivy file, Gradle provides seamless integration with a range of popular build tools.
- Integration with existing dependency management infrastructure: Gradle is compatible with both Maven and Ivy repositories. If you use Archiva, Nexus, or Artifactory, Gradle is 100% compatible with all repository formats.

With hundreds of thousands of interdependent open source components each with a range of versions and incompatibilities, dependency management has a habit of causing problems as builds grow in complexity. When a build's dependency tree becomes unwieldy, your build tool shouldn't force you to adopt a single, inflexible approach to dependency management. A proper build system has to be designed to be flexible, and Gradle can handle any situation.

25.1.1. Flexible dependency management for migrations

Dependency management can be particularly challenging during a migration from one build system to another. If you are migrating from a tool like Ant or Maven to Gradle, you may be faced with some difficult situations. For example, one common pattern is an Ant project with version-less jar files stored in the filesystem. Other build systems require a wholesale replacement of this approach before migrating. With Gradle, you can adapt your new build to any existing source of dependencies or dependency metadata. This makes incremental migration to Gradle much easier than the alternative. On most large projects, build migrations and any change to development process is incremental because most organizations can't afford to stop everything and migrate to a build tool's idea of dependency management.

Even if your project is using a custom dependency management system or something like an Eclipse .classpath file as master data for dependency management, it is very easy to write a Gradle plugin to use this data in Gradle. For migration purposes this is a common technique with Gradle. (But, once you've migrated, it might be a good idea to move away from a .classpath file and use Gradle's dependency management features directly.)

25.1.2. Dependency management and Java

It is ironic that in a language known for its rich library of open source components that Java has no concept of libraries or versions. In Java, there is no standard way to tell the JVM that you are using version 3.0.5 of Hibernate, and there is no standard way to say that `foo-1.0.jar` depends on `bar-2.0.jar`. This has led to external solutions often based on build tools. The most popular ones at the moment are Maven and Ivy. While Maven provides a complete build system, Ivy focuses solely on dependency management.

Both tools rely on descriptor XML files, which contain 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. Both have emerged as standards for solving dependency conflicts, and while Gradle originally used Ivy under the hood for its dependency management. Gradle has replaced this direct dependency on Ivy with a native Gradle dependency resolution engine which supports a range of approaches to dependency resolution including both POM and Ivy descriptor files.

25.2. Dependency Management Best Practices

While Gradle has strong opinions on dependency management, the tool gives you a choice between two options: follow recommended best practices or support any kind of pattern you can think of. This section outlines the Gradle project's recommended best practices for managing dependencies.

No matter what the language, proper dependency management is important for every project. From a complex enterprise application written in Java depending on hundreds of open source libraries to the simplest Clojure application depending on a handful of libraries, approaches to dependency management vary widely and can depend on the target technology, the method of application deployment, and the nature of the project. Projects bundled as reusable libraries may have different requirements than enterprise applications integrated into much larger systems of software and infrastructure. Despite this wide variation of requirements, the Gradle project recommends that all projects follow this set of core rules:

25.2.1. Put the Version in the Filename (Version the jar)

The version of a library must be part of the filename. While the version of a jar is usually in the Manifest file, it isn't readily apparent when you are inspecting a project. If someone asks you to look at a collection of 20 jar files, which would you prefer? A collection of files with names like `commons-beanutils-1.3.jar` or a collection of files with names like `spring.jar`? If dependencies have file names with version numbers you can quickly identify the versions of your dependencies.

If versions are unclear you can introduce subtle bugs which are very hard to find. For example there might be a project which uses Hibernate 2.5. Think about a developer who decides to install version 3.0.5 of Hibernate on her machine to fix a critical security bug but forgets to notify others in the team of this change. She may address the security bug successfully, but she also may have introduced subtle bugs into a codebase that was using a now-deprecated feature from Hibernate. Weeks later there is an exception on the integration machine which can't be reproduced on anyone's machine. Multiple developers then spend days on this issue only finally realising that the error would have been easy to uncover if they knew that Hibernate had been upgraded from 2.5 to 3.0.5.

Versions in jar names increase the expressiveness of your project and make them easier to maintain. This practice also reduces the potential for error.

25.2.2. Manage transitive dependencies

Transitive dependency management is a technique that enables your project to depend on libraries which, in turn, depend on other libraries. This recursive pattern of transitive dependencies results in a tree of dependencies including your project's first-level dependencies, second-level dependencies, and so on. If you don't model your dependencies as a hierarchical tree of first-level and second-level dependencies it is very easy to quickly lose control over an assembled mess of unstructured dependencies. Consider the Gradle project itself, while Gradle only has a few direct, first-level dependencies, when Gradle is compiled it needs more than one hundred dependencies on the classpath. On a far larger scale, Enterprise projects using Spring, Hibernate, and other libraries, alongside hundreds or thousands of internal projects, can result in very large dependency trees.

When these large dependency trees need to change, you'll often have to solve some dependency version conflicts. Say one open source library needs one version of a logging library and another uses an alternative version. Gradle and other build tools all have the ability to resolve conflicts, but what differentiates Gradle is the control it gives you over transitive dependencies and conflict resolution.

While you could try to manage this problem manually, you will quickly find that this approach doesn't scale. If you want to get rid of a first level dependency you really can't be sure which other jars you should remove. A dependency of a first level dependency might also be a first level dependency itself, or it might be a transitive dependency of yet another first level dependency. If you try to manage transitive dependencies yourself, the end of the story is that your build becomes brittle: no one dares to change your dependencies because the risk of breaking the build is too high. The project classpath becomes a complete mess, and, if a classpath problem arises, hell on earth invites you for a ride.

NOTE: In one project, we found a mystery LDAP related jar in the classpath. No code referenced this jar and there was no connection to the project. No one could figure out what the jar was for, until it was removed from the build and the application suffered massive performance problems whenever it

attempted to authenticate to LDAP. This mystery jar was a necessary transitive, fourth-level dependency that was easy to miss because no one had bothered to use managed transitive dependencies.

Gradle offers you different ways to express first-level and transitive dependencies. With Gradle you can mix and match approaches; for example, you could store your jars in an SCM without XML descriptor files and still use transitive dependency management.

25.2.3. Resolve version conflicts

Conflicting versions of the same jar should be detected and either resolved or cause an exception. If you don't use transitive dependency management, version conflicts are undetected and the often accidental order of the classpath will determine what version of a dependency will win. On a large project with many developers changing dependencies, successful builds will be few and far between as the order of dependencies may directly affect whether a build succeeds or fails (or whether a bug appears or disappears in production).

If you haven't had to deal with the curse of conflicting versions of jars on a classpath, here is a small anecdote of the fun that awaits you. In a large project with 30 submodules, adding a dependency to a subproject changed the order of a classpath, swapping Spring 2.5 for an older 2.4 version. While the build continued to work, developers were starting to notice all sorts of surprising (and surprisingly awful) bugs in production. Worse yet, this unintentional downgrade of Spring introduced several security vulnerabilities into the system, which now required a full security audit throughout the organization.

In short, version conflicts are bad, and you should manage your transitive dependencies to avoid them. You might also want to learn where conflicting versions are used and consolidate on a particular version of a dependency across your organization. With a good conflict reporting tool like Gradle, that information can be used to communicate with the entire organization and standardize on a single version. *If you think version conflicts don't happen to you, think again.* It is very common for different first-level dependencies to rely on a range of different overlapping versions for other dependencies, and the JVM doesn't yet offer an easy way to have different versions of the same jar in the classpath (see Section 25.1.2, “Dependency management and Java”).

Gradle offers the following conflict resolution strategies:

- *Newest*: The newest version of the dependency is used. This is Gradle's default strategy, and is often an appropriate choice as long as versions are backwards-compatible.
- *Fail*: A version conflict results in a build failure. This strategy requires all version conflicts to be resolved explicitly in the build script. See `ResolutionStrategy` for details on how to explicitly choose a particular version.

While the strategies introduced above are usually enough to solve most conflicts, Gradle provides more fine-grained mechanisms to resolve version conflicts:

- Configuring a first level dependency as *forced*. This approach is useful if the dependency in conflict is already a first level dependency. See examples in `DependencyHandler`.
- Configuring any dependency (transitive or not) as *forced*. This approach is useful if the dependency in conflict is a transitive dependency. It also can be used to force versions of first level dependencies. See

examples in `ResolutionStrategy`

- Configuring dependency resolution to *prefer modules that are part of your build* (transitive or not). This approach is useful if your build contains custom forks of modules (as part of Chapter 26, *Multi-project Builds* or as include in Chapter 10, *Composite builds*). See examples in `ResolutionStrategy`.
- Dependency resolve rules are an incubating feature introduced in Gradle 1.4 which give you fine-grained control over the version selected for a particular dependency.

To deal with problems due to version conflicts, reports with dependency graphs are also very helpful. Such reports are another feature of dependency management.

25.2.4. Use Dynamic Versions and Changing Modules

There are many situations when you want to use the latest version of a particular dependency, or the latest in a range of versions. This can be a requirement during development, or you may be developing a library that is designed to work with a range of dependency versions. You can easily depend on these constantly changing dependencies by using a *dynamic version*. A dynamic version can be either a version range (e.g. `2.+`) or it can be a placeholder for the latest version available (e.g. `latest.integration`).

Alternatively, sometimes the module you request can change over time, even for the same version. An example of this type of *changing module* is a Maven SNAPSHOT module, which always points at the latest artifact published. In other words, a standard Maven snapshot is a module that never stands still so to speak, it is a “changing module”.

The main difference between a *dynamic version* and a *changing module* is that when you resolve a *dynamic version*, you'll get the real, static version as the module name. When you resolve a *changing module*, the artifacts are named using the version you requested, but the underlying artifacts may change over time.

By default, Gradle caches dynamic versions and changing modules for 24 hours. You can override the default cache modes using command line options. You can change the cache expiry times in your build using the resolution strategy (see Section 25.9.3, “Fine-tuned control over dependency caching”).

25.3. Dependency configurations

In Gradle dependencies are grouped into configurations. Configurations have a name, a number of other properties, and they can extend each other. Many Gradle plugins add pre-defined configurations to your project. The Java plugin, for example, adds some configurations to represent the various classpaths it needs. see Section 47.5, “Dependency management” for details. Of course you can 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).

A project's configurations are managed by a `configurations` object. The closure you pass to the `configurations` object is applied against its API. To learn more about this API have a look at `ConfigurationContainer`.

To define a configuration:

Example 25.1. Definition of a configuration

build.gradle

```
configurations {  
    compile  
}
```

To access a configuration:

Example 25.2. Accessing a configuration

build.gradle

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

To configure a configuration:

Example 25.3. Configuration of a configuration

build.gradle

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

25.4. How to declare your dependencies

There are several different types of dependencies that you can declare:

Table 25.1. Dependency types

Type	Description
External module dependency	A dependency on an external module in some repository.
Project dependency	A dependency on another project in the same build.
File dependency	A dependency on a set of files on the local filesystem.
Client module dependency	A dependency on an external module, where the artifacts are located in some repository but the module meta-data is specified by the local build. You use this kind of dependency when you want to override the meta-data for the module.
Gradle API dependency	A dependency on the API of the current Gradle version. You use this kind of dependency when you are developing custom Gradle plugins and task types.
Local Groovy dependency	A dependency on the Groovy version used by the current Gradle version. You use this kind of dependency when you are developing custom Gradle plugins and task types.

25.4.1. External module dependencies

External module dependencies are the most common dependencies. They refer to a module in an external repository.

Example 25.4. 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
    }
}
```

See the `DependencyHandler` class in the API documentation for more examples and a complete reference.

Gradle provides different notations for module dependencies. There is a string notation and a map notation. A module dependency has an API which allows 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 of 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 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`) 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 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.

Depending on modules with multiple artifacts

As mentioned earlier, a Maven module has only one artifact. Hence, when your project depends on a Maven module, it's obvious what its artifact is. With Gradle or Ivy, the case is different. Ivy's dependency descriptor (`ivy.xml`) can declare multiple artifacts. For more information, see the Ivy reference for `ivy.xml`. In Gradle, when you declare a dependency on an Ivy module, you actually declare a dependency on the `default` configuration of that module. So the actual set of artifacts (typically jars) you depend on is the set of artifacts that are associated with the `default` configuration of that module. Here are some situations where this matters:

- The `default` configuration of a module contains undesired artifacts. Rather than depending on the whole configuration, a dependency on just the desired artifacts is declared.
- The desired artifact belongs to a configuration other than `default`. That configuration is explicitly named as part of the dependency declaration.

There are other situations where it is necessary to fine-tune dependency declarations. Please see the `DependencyHandler` class in the API documentation for examples and a complete reference for declaring dependencies.

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.^[11] 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 25.5. Artifact only notation

build.gradle

```
dependencies {
    runtime "org.groovy:groovy:2.2.0@jar"
    runtime group: 'org.groovy', name: 'groovy', version: '2.2.0', 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.

Classifiers

The Maven dependency management has the notion of classifiers. ^[12] Gradle supports this. To retrieve classified dependencies from a Maven repository you can write:

Example 25.6. 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',
```

As can be seen in the first line above, classifiers can be used together with the artifact only notation.

It is easy to iterate over the dependency artifacts of a configuration:

Example 25.7. Iterating over a configuration

build.gradle

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

Output of `gradle -q listJars`

```
> gradle -q listJars
hibernate-core-3.6.7.Final.jar
antlr-2.7.6.jar
commons-collections-3.1.jar
dom4j-1.6.1.jar
hibernate-commons-annotations-3.2.0.Final.jar
hibernate-jpa-2.0-api-1.0.1.Final.jar
jta-1.1.jar
slf4j-api-1.6.1.jar
```

25.4.2. Client module dependencies

Client module dependencies allow you to declare *transitive* dependencies directly in the build script. They are a replacement for a module descriptor in an external repository.

Example 25.8. Client module dependencies - transitive dependencies

build.gradle

```
dependencies {
    runtime module("org.codehaus.groovy:groovy:2.4.7") {
        dependency("commons-cli:commons-cli:1.0") {
            transitive = false
        }
        module(group: 'org.apache.ant', name: 'ant', version: '1.9.6') {
            dependencies "org.apache.ant:ant-launcher:1.9.6@jar",
                        "org.apache.ant:ant-junit:1.9.6"
        }
    }
}
```

This declares a dependency 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. Also look at the API documentation for the `ClientModule` class.

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.

25.4.3. 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 25.9. Project dependencies

build.gradle

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

For more information see the API documentation for `ProjectDependency`.

Multi-project builds are discussed in Chapter 26, *Multi-project Builds*.

25.4.4. 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 25.10. File dependencies

build.gradle

```
dependencies {  
    runtime files('libs/a.jar', 'libs/b.jar')  
    runtime fileTree(dir: 'libs', include: '*.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 25.11. Generated file dependencies

build.gradle

```
dependencies {  
    compile files("$buildDir/classes") {  
        builtBy 'compile'  
    }  
}  
  
task compile {  
    doLast {  
        println 'compiling classes'  
    }  
}  
  
task list(dependsOn: configurations.compile) {  
    doLast {  
        println "classpath = ${configurations.compile.collect { File file -> file.name } }"  
    }  
}
```

Output of **gradle -q list**

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

25.4.5. Gradle API Dependency

You can declare a dependency on the API of the current version of Gradle by using the `DependencyHandler.gradleApi()` method. This is useful when you are developing custom Gradle tasks or plugins.

Example 25.12. Gradle API dependencies

build.gradle

```
dependencies {
    compile gradleApi()
}
```

25.4.6. Local Groovy Dependency

You can declare a dependency on the Groovy that is distributed with Gradle by using the `DependencyHandler.localGroovy()` method. This is useful when you are developing custom Gradle tasks or plugins in Groovy.

Example 25.13. Gradle's Groovy dependencies

build.gradle

```
dependencies {
    compile localGroovy()
}
```

25.4.7. Excluding transitive dependencies

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

Example 25.14. 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. Also look at the API documentation of the `Dependency` and `Configuration` classes.

Not every transitive dependency can be excluded - some transitive dependencies might be essential for correct runtime behavior of the application. Generally, one can exclude transitive dependencies that are either not required by runtime or that are guaranteed to be available on the target environment/platform.

Should you exclude per-dependency or per-configuration? It turns out that in the majority of cases you want to use the per-configuration exclusion. Here are some typical reasons why one might want to exclude a transitive dependency. Bear in mind that for some of these use cases there are better solutions than exclusions!

- The dependency is undesired due to licensing reasons.
- The dependency is not available in any remote repositories.
- The dependency is not needed for runtime.
- The dependency has a version that conflicts with a desired version. For that use case please refer to Section 25.2.3, “Resolve version conflicts” and the documentation on `ResolutionStrategy` for a potentially better solution to the problem.

Basically, in most of the cases excluding the transitive dependency should be done per configuration. This way the dependency declaration is more explicit. It is also more accurate because a per-dependency exclude rule does not guarantee the given transitive dependency does not show up in the configuration. For example, some other dependency, which does not have any exclude rules, might pull in that unwanted transitive dependency.

Other examples of dependency exclusions can be found in the reference for the `ModuleDependency` or `DependencyHandler` classes.

25.4.8. Optional attributes

All attributes for a dependency are optional, except the name. Which attributes are required for actually finding dependencies in the repository will depend on the repository type. See Section 25.6, “Repositories”. For example, if you work 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 25.15. Optional attributes of dependencies

build.gradle

```
dependencies {
    runtime ":junit:4.12", ":testng"
    runtime name: 'testng'
}
```

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

Example 25.16. Collections and arrays of dependencies

build.gradle

```
List groovy = ["org.codehaus.groovy:groovy-all:2.4.7@jar",
               "commons-cli:commons-cli:1.0@jar",
               "org.apache.ant:ant:1.9.6@jar"]
List hibernate = ['org.hibernate:hibernate:3.0.5@jar',
                  'somegroup:someorg:1.0@jar']
dependencies {
    runtime groovy, hibernate
}
```

25.4.9. 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 possibility 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 25.17. Dependency configurations

build.gradle

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

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

Example 25.18. Dependency configurations for project

build.gradle

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

25.4.10. Dependency reports

You can generate dependency reports from the command line (see Section 4.7.4, “Listing project dependencies”). With the help of the Project report plugin (see Chapter 29, *The Project Report Plugin*) such a report can be created by your build.

Since Gradle 1.2 there is also a new programmatic API to access the resolved dependency information. The dependency reports (see the previous paragraph) are using this API under the covers. The API lets you walk the resolved dependency graph and provides information about the dependencies. In future releases the API will grow to provide more information about the resolution result. For more information about the API please refer to the Javadocs on `ResolvableDependencies.getResolutionResult()`. Potential usages of the `ResolutionResult` API:

- Creation of advanced dependency reports tailored to your use case.
- Enabling the build logic to make decisions based on the content of the dependency graph.

25.5. Working with dependencies

For the examples below we have the following dependencies setup:

Example 25.19. Configuration.copy

build.gradle

```
configurations {
    sealife
    alllife
}

dependencies {
    sealife "sea.mammals:orca:1.0", "sea.fish:shark:1.0", "sea.fish:tuna:1.0"
    alllife configurations.sealife
    alllife "air.birds:albatross: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 25.20. Accessing declared dependencies

build.gradle

```
task dependencies {
    doLast {
        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 { dep -> println dep.name }
    }
}
```

Output of **gradle -q dependencies**

```
> gradle -q dependencies
albatross
```

```
albatross
orca
shark
tuna
```

```
albatross
shark
tuna
```

The `dependencies` task returns only the dependencies belonging explicitly to the configuration. The `allDependencies` task includes the dependencies from extended configurations.

To get the library files of the configuration dependencies you can do:

Example 25.21. Configuration.files

build.gradle

```
task allFiles {
    doLast {
        configurations.sealife.files.each { file ->
            println file.name
        }
    }
}
```

Output of `gradle -q allFiles`

```
> gradle -q allFiles
orca-1.0.jar
shark-1.0.jar
tuna-1.0.jar
herring-1.0.jar
seal-2.0.jar
```

Sometimes you want the library files of a subset of the configuration dependencies (e.g. of a single dependency).

Example 25.22. Configuration.files with spec

build.gradle

```
task files {
    doLast {
        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 25.23. Configuration.copy

build.gradle

```
task copy {
    doLast {
        configurations.alllife.copyRecursive { dep -> dep.name != 'orca' }
        .allDependencies.each { dep -> println dep.name }
        println()
        configurations.alllife.copy().allDependencies
        .each { dep -> println dep.name }
    }
}
```

Output of **gradle -q copy**

```
> gradle -q copy
albatross
shark
tuna

albatross
```

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 25.24. Configuration.copy vs. Configuration.files

build.gradle

```
task copyVsFiles {
    doLast {
        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 API documentation: [Configuration](#).

25.6. Repositories

Gradle repository management, based on Apache Ivy, gives you a lot of freedom regarding repository layout and retrieval policies. Additionally Gradle provides various convenience method to add pre-configured repositories.

You may configure any number of repositories, each of which is treated independently by Gradle. If Gradle finds a module descriptor in a particular repository, it will attempt to download all of the artifacts for that module from *the same repository*. Although module meta-data and module artifacts must be located in the same repository, it is possible to compose a single repository of multiple URLs, giving multiple locations to search for meta-data files and jar files.

There are several different types of repositories you can declare:

Table 25.2. Repository types

Type	Description
Maven central repository	A pre-configured repository that looks for dependencies in Maven Central.
Maven JCenter repository	A pre-configured repository that looks for dependencies in Bintray's JCenter.
Maven local repository	A pre-configured repository that looks for dependencies in the local Maven repository.
Maven repository	A Maven repository. Can be located on the local filesystem or at some remote location.
Ivy repository	An Ivy repository. Can be located on the local filesystem or at some remote location.
Flat directory repository	A simple repository on the local filesystem. Does not support any meta-data formats.

25.6.1. Maven central repository

To add the central Maven 2 repository (<https://repo1.maven.org/maven2>) simply add this to your build script:

Example 25.25. Adding central Maven repository

build.gradle

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

Now Gradle will look for your dependencies in this repository.

25.6.2. Maven JCenter repository

Bintray's JCenter is an up-to-date collection of all popular Maven OSS artifacts, including artifacts published directly to Bintray.

To add the JCenter Maven repository (<https://jcenter.bintray.com>) simply add this to your build script:

Example 25.26. Adding Bintray's JCenter Maven repository

build.gradle

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

Now Gradle will look for your dependencies in the JCenter repository. `jcenter()` uses HTTPS to connect to the repository. If you want to use HTTP you can configure `jcenter()`:

Example 25.27. Using Bintray's JCenter with HTTP

build.gradle

```
repositories {  
    jcenter {  
        url "http://jcenter.bintray.com/"  
    }  
}
```

25.6.3. Local Maven repository

To use the local Maven cache as a repository you can do:

Example 25.28. Adding the local Maven cache as a repository

build.gradle

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

Gradle uses the same logic as Maven to identify the location of your local Maven cache. If a local repository location is defined in a `settings.xml`, this location will be used. The `settings.xml` in `USER_HOME/.m2` takes precedence over the `settings.xml` in `M2_HOME/conf`. If no `settings.xml` is available,

Gradle uses the default location `USER_HOME/.m2/repository`.

25.6.4. Maven repositories

For adding a custom Maven repository you can do:

Example 25.29. Adding custom Maven repository

build.gradle

```
repositories {
    maven {
        url "http://repo.mycompany.com/maven2"
    }
}
```

Sometimes a repository will have the POMs published to one location, and the JARs and other artifacts published at another location. To define such a repository, you can do:

Example 25.30. Adding additional Maven repositories for JAR files

build.gradle

```
repositories {
    maven {
        // Look for POMs and artifacts, such as JARs, here
        url "http://repo2.mycompany.com/maven2"
        // Look for artifacts here if not found at the above location
        artifactUrls "http://repo.mycompany.com/jars"
        artifactUrls "http://repo.mycompany.com/jars2"
    }
}
```

Gradle will look at the first URL for the POM and the JAR. If the JAR can't be found there, the artifact URLs are used to look for JARs.

Accessing password protected Maven repositories

To access a Maven repository which uses basic authentication, you specify the username and password to use when you define the repository:

Example 25.31. Accessing password protected Maven repository

build.gradle

```
repositories {
    maven {
        credentials {
            username 'user'
            password 'password'
        }
        url "http://repo.mycompany.com/maven2"
    }
}
```

It is advisable to keep your username and password in `gradle.properties` rather than directly in the build file.

25.6.5. Flat directory repository

If you want to use a (flat) filesystem directory as a repository, simply type:

Example 25.32. Flat repository resolver

build.gradle

```
repositories {
    flatDir {
        dirs 'lib'
    }
    flatDir {
        dirs 'lib1', 'lib2'
    }
}
```

This adds repositories which look into one or more directories for finding dependencies. Note that this type of repository does not support any meta-data formats like Ivy XML or Maven POM files. Instead, Gradle will dynamically generate a module descriptor (without any dependency information) based on the presence of artifacts. However, as Gradle prefers to use modules whose descriptor has been created from real meta-data rather than being generated, flat directory repositories cannot be used to override artifacts with real meta-data from other repositories. So, for example, if Gradle finds only `jmxri-1.2.1.jar` in a flat directory repository, but `jmxri-1.2.1.pom` in another repository that supports meta-data, it will use the second repository to provide the module. For the use case of overriding remote artifacts with local ones consider using an Ivy or Maven repository instead whose URL points to a local directory. If you only work with flat directory repositories you don't need to set all attributes of a dependency. See Section 25.4.8, “Optional attributes”.

25.6.6. Ivy repositories

Defining an Ivy repository with a standard layout

Example 25.33. Ivy repository

build.gradle

```
repositories {
    ivy {
        url "http://repo.mycompany.com/repo"
    }
}
```

Defining a named layout for an Ivy repository

You can specify that your repository conforms to the Ivy or Maven default layout by using a named layout.

Example 25.34. Ivy repository with named layout

build.gradle

```
repositories {
    ivy {
        url "http://repo.mycompany.com/repo"
        layout "maven"
    }
}
```

Valid named layout values are 'gradle' (the default), 'maven', 'ivy' and 'pattern'. See `IvyArtifactRepository.layout(java.lang.String, groovy.lang.Closure)` in the API documentation for details of these named layouts.

Defining custom pattern layout for an Ivy repository

To define an Ivy repository with a non-standard layout, you can define a 'pattern' layout for the repository:

Example 25.35. Ivy repository with pattern layout

build.gradle

```
repositories {
    ivy {
        url "http://repo.mycompany.com/repo"
        layout "pattern", {
            artifact "[module]/[revision]/[type]/[artifact].[ext]"
        }
    }
}
```

To define an Ivy repository which fetches Ivy files and artifacts from different locations, you can define separate patterns to use to locate the Ivy files and artifacts:

Each `artifact` or `ivy` specified for a repository adds an *additional* pattern to use. The patterns are used in the order that they are defined.

Example 25.36. Ivy repository with multiple custom patterns

build.gradle

```
repositories {
    ivy {
        url "http://repo.mycompany.com/repo"
        layout "pattern", {
            artifact "3rd-party-artifacts/[organisation]/[module]/[revision]/[artifact].[ext]"
            artifact "company-artifacts/[organisation]/[module]/[revision]/[artifact].[ext]"
            ivy "ivy-files/[organisation]/[module]/[revision]/ivy.xml"
        }
    }
}
```

Optionally, a repository with pattern layout can have its 'organisation' part laid out in Maven style, with forward slashes replacing dots as separators. For example, the organisation `my.company` would then be

represented as `my/company`.

Example 25.37. Ivy repository with Maven compatible layout

build.gradle

```
repositories {
    ivy {
        url "http://repo.mycompany.com/repo"
        layout "pattern", {
            artifact "[organisation]/[module]/[revision]/[artifact]-[revision].[ext]"
            m2compatible = true
        }
    }
}
```

Accessing password protected Ivy repositories

To access an Ivy repository which uses basic authentication, you specify the username and password to use when you define the repository:

Example 25.38. Ivy repository

build.gradle

```
repositories {
    ivy {
        url 'http://repo.mycompany.com'
        credentials {
            username 'user'
            password 'password'
        }
    }
}
```

25.6.7. Supported repository transport protocols

Maven and Ivy repositories support the use of various transport protocols. At the moment the following protocols are supported:

Table 25.3. Repository transport protocols

Type	Credential types
file	none
http	username/password
https	username/password
sftp	username/password
s3	access key/secret key/session token or Environment variables

To define a repository use the `repositories` configuration block. Within the `repositories` closure, a Maven repository is declared with `maven`. An Ivy repository is declared with `ivy`. The transport protocol

is part of the URL definition for a repository. The following build script demonstrates how to create a HTTP-based Maven and Ivy repository:

Example 25.39. Declaring a Maven and Ivy repository

build.gradle

```
repositories {
    maven {
        url "http://repo.mycompany.com/maven2"
    }

    ivy {
        url "http://repo.mycompany.com/repo"
    }
}
```

If authentication is required for a repository, the relevant credentials can be provided. The following example shows how to provide username/password-based authentication for SFTP repositories:

Example 25.40. Providing credentials to a Maven and Ivy repository

build.gradle

```
repositories {
    maven {
        url "sftp://repo.mycompany.com:22/maven2"
        credentials {
            username 'user'
            password 'password'
        }
    }

    ivy {
        url "sftp://repo.mycompany.com:22/repo"
        credentials {
            username 'user'
            password 'password'
        }
    }
}
```

When using an AWS S3 backed repository you need to authenticate using `AwsCredentials`, providing access-key and a private-key. The following example shows how to declare a S3 backed repository and providing AWS credentials:

Example 25.41. Declaring a S3 backed Maven and Ivy repository

build.gradle

```
repositories {
    maven {
        url "s3://myCompanyBucket/maven2"
        credentials(AwsCredentials) {
            accessKey "someKey"
            secretKey "someSecret"
            // optional
            sessionToken "someSTSToken"
        }
    }

    ivy {
        url "s3://myCompanyBucket/ivyrepo"
        credentials(AwsCredentials) {
            accessKey "someKey"
            secretKey "someSecret"
            // optional
            sessionToken "someSTSToken"
        }
    }
}
```

You can also delegate all credentials to the AWS sdk by using the `AwsImAuthentication`. The following example shows how :

Example 25.42. Declaring a S3 backed Maven and Ivy repository using IAM

build.gradle

```
repositories {
    maven {
        url "s3://myCompanyBucket/maven2"
        authentication {
            awsIm(AwsImAuthentication) // load from EC2 role or env var
        }
    }

    ivy {
        url "s3://myCompanyBucket/ivyrepo"
        authentication {
            awsIm(AwsImAuthentication)
        }
    }
}
```

S3 configuration properties

The following system properties can be used to configure the interactions with s3 repositories:

Table 25.4. S3 Configuration Properties

Property	Description
<code>org.gradle.s3.endpoint</code>	Used to override the AWS S3 endpoint when using a non AWS, S3 API compatible, storage service.
<code>org.gradle.s3.maxErrorRetry</code>	Specifies the maximum number of times to retry a request in the event that the S3 server responds with a HTTP 5xx status code. When not specified a default value of 3 is used.

S3 URL formats

S3 URL's are 'virtual-hosted-style' and must be in the following format `s3://<bucketName>[.<regionSpe`

e.g. `s3://myBucket.s3.eu-central-1.amazonaws.com/maven/release`

- `myBucket` is the AWS S3 bucket name.
- `s3.eu-central-1.amazonaws.com` is the *optional* region specific endpoint.
- `/maven/release` is the AWS S3 key (unique identifier for an object within a bucket)

S3 proxy settings

A proxy for S3 can be configured using the following system properties:

- `https.proxyHost`
- `https.proxyPort`
- `https.proxyUser`
- `https.proxyPassword`
- `http.nonProxyHosts`

If the '`org.gradle.s3.endpoint`' property has been specified with a http (not https) URI the following system proxy settings can be used:

- `http.proxyHost`
- `http.proxyPort`
- `http.proxyUser`
- `http.proxyPassword`
- `http.nonProxyHosts`

AWS S3 V4 Signatures (AWS4-HMAC-SHA256)

Some of the AWS S3 regions (eu-central-1 - Frankfurt) require that all HTTP requests are signed in accordance with AWS's signature version 4. It is recommended to specify S3 URL's containing the region specific endpoint when using buckets that require V4 signatures. e.g. `s3://somebucket.s3.eu-central-`

NOTE: When a region-specific endpoint is not specified for buckets requiring V4 Signatures, Gradle will use the default AWS region (us-east-1) and the following warning will appear on the console:
Attempting to re-send the request to with AWS V4 authentication. To avoid this warning in the

future, please use region-specific endpoint to access buckets located in regions that require V4 signing.

Failing to specify the region-specific endpoint for buckets requiring V4 signatures means:

- 3 round-trips to AWS, as opposed to one, for every file upload and download
- Depending on location - increased network latencies and slower builds.
- Increased likelihood of transmission failures.

Configuring HTTP authentication schemes

When configuring a repository using HTTP or HTTPS transport protocols, multiple authentication schemes are available. By default, Gradle will attempt to use all schemes that are supported by the Apache HttpClient library, documented here. In some cases, it may be preferable to explicitly specify which authentication schemes should be used when exchanging credentials with a remote server. When explicitly declared, only those schemes are used when authenticating to a remote repository. The following example show how to configure a repository to use only digest authentication:

Example 25.43. Configure repository to use only digest authentication

build.gradle

```
repositories {
    maven {
        url 'https://repo.mycompany.com/maven2'
        credentials {
            username 'user'
            password 'password'
        }
        authentication {
            digest(DigestAuthentication)
        }
    }
}
```

Currently supported authentication schemes are:

Table 25.5. Authentication schemes

Type	Description
BasicAuthentication	Basic access authentication over HTTP. When using this scheme, credentials are sent preemptively.
DigestAuthentication	Digest access authentication over HTTP.

Using preemptive authentication

Gradle's default behavior is to only submit credentials when a server responds with an authentication challenge in the form of a HTTP 401 response. In some cases, the server will respond with a different code (ex. for repositories hosted on GitHub a 404 is returned) causing dependency resolution to fail. To get around this behavior, credentials may be sent to the server preemptively. To enable preemptive authentication simply configure your repository to explicitly use the `BasicAuthentication` scheme:

Example 25.44. Configure repository to use preemptive authentication

build.gradle

```
repositories {
    maven {
        url 'https://repo.mycompany.com/maven2'
        credentials {
            username 'user'
            password 'password'
        }
        authentication {
            basic(BasicAuthentication)
        }
    }
}
```

25.6.8. Working with repositories

To access a repository:

Example 25.45. Accessing a repository

build.gradle

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

To configure a repository:

Example 25.46. Configuration of a repository

build.gradle

```
repositories {
    flatDir {
        name 'localRepository'
    }
}
repositories {
    localRepository {
        dirs 'lib'
    }
}
repositories.localRepository {
    dirs 'lib'
}
```

25.6.9. More about Ivy resolvers

Gradle is extremely flexible regarding repositories:

- There are many options for the protocol to communicate with the repository (e.g. filesystem, http, ssh, sftp ...)
- The protocol sftp currently only supports username/password-based authentication.
- 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: ^[13]

```
// Maven2 layout (if a repository is marked as Maven2 compatible, the organization is split into subfolders)
someroot/[organisation]/[module]/[revision]/[module]-[revision].[ext]

// Typical layout for an Ivy repository (the organization is not split into subfolders)
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 25.47. Definition of a custom repository

build.gradle

```
repositories {
    ivy {
        ivyPattern "$projectDir/repo/[organisation]/[module]-ivy-[revision].xml"
        artifactPattern "$projectDir/repo/[organisation]/[module]-[revision](-[classifier])-[artifact].[ext]"
    }
}
```

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.

25.7. How dependency resolution works

Gradle takes your dependency declarations and repository definitions and attempts to download all of your dependencies by a process called *dependency resolution*. Below is a brief outline of how this process works.

- Given a required dependency, Gradle first attempts to resolve the *module* for that dependency. Each repository is inspected in order, searching first for a *module descriptor* file (POM or Ivy file) that indicates the presence of that module. If no module descriptor is found, Gradle will search for the presence of the primary *module artifact* file indicating that the module exists in the repository.
 - If the dependency is declared as a dynamic version (like 1. +), Gradle will resolve this to the newest available static version (like 1. 2) in the repository. For Maven repositories, this is done using the maven-file, while for Ivy repositories this is done by directory listing.
 - If the module descriptor is a POM file that has a parent POM declared, Gradle will recursively attempt to resolve each of the parent modules for the POM.
 - Once each repository has been inspected for the module, Gradle will choose the 'best' one to use. This is done using the following criteria:
 - For a dynamic version, a 'higher' static version is preferred over a 'lower' version.
 - Modules declared by a module descriptor file (Ivy or POM file) are preferred over modules that have an artifact file only.
 - Modules from earlier repositories are preferred over modules in later repositories.
- When the dependency is declared by a static version and a module descriptor file is found in a repository, there is no need to continue searching later repositories and the remainder of the process is short-circuited.
- All of the artifacts for the module are then requested from the *same repository* that was chosen in the process above.

25.8. Fine-tuning the dependency resolution process

In most cases, Gradle's default dependency management will resolve the dependencies that you want in your build. In some cases, however, it can be necessary to tweak dependency resolution to ensure that your build receives exactly the right dependencies.

There are a number of ways that you can influence how Gradle resolves dependencies.

25.8.1. Forcing a particular module version

Forcing a module version tells Gradle to always use a specific version for given dependency (transitive or not), overriding any version specified in a published module descriptor. This can be very useful when tackling version conflicts - for more information see Section 25.2.3, "Resolve version conflicts".

Force versions can also be used to deal with rogue metadata of transitive dependencies. If a transitive

dependency has poor quality metadata that leads to problems at dependency resolution time, you can force Gradle to use a newer, fixed version of this dependency. For an example, see the `ResolutionStrategy` class in the API documentation. Note that 'dependency resolve rules' (outlined below) provide a more powerful mechanism for replacing a broken module dependency. See the section called “Blacklisting a particular version with a replacement”.

25.8.2. Preferring modules that are part of the your build

Preferring project modules tells Gradle to use the version of a module that is part of the build itself (as part of Chapter 26, *Multi-project Builds* or as includes in Chapter 10, *Composite builds*). This allows the easy inclusion of an individual fork (e.g. containing a bugfix) of a module - for more information see Section 25.2.3, “Resolve version conflicts”.

25.8.3. Using dependency resolve rules

A dependency resolve rule is executed for each resolved dependency, and offers a powerful api for manipulating a requested dependency prior to that dependency being resolved. This feature is incubating, but currently offers the ability to change the group, name and/or version of a requested dependency, allowing a dependency to be substituted with a completely different module during resolution.

Dependency resolve rules provide a very powerful way to control the dependency resolution process, and can be used to implement all sorts of advanced patterns in dependency management. Some of these patterns are outlined below. For more information and code samples see the `ResolutionStrategy` class in the API documentation.

Modelling releasable units

Often an organisation publishes a set of libraries with a single version; where the libraries are built, tested and published together. These libraries form a 'releasable unit', designed and intended to be used as a whole. It does not make sense to use libraries from different releasable units together.

But it is easy for transitive dependency resolution to violate this contract. For example:

- `module-a` depends on `releasable-unit:part-one:1.0`
- `module-b` depends on `releasable-unit:part-two:1.1`

A build depending on both `module-a` and `module-b` will obtain different versions of libraries within the releasable unit.

Dependency resolve rules give you the power to enforce releasable units in your build. Imagine a releasable unit defined by all libraries that have 'org.gradle' group. We can force all of these libraries to use a consistent version:

Example 25.48. Forcing consistent version for a group of libraries

build.gradle

```
configurations.all {
    resolutionStrategy.eachDependency { DependencyResolveDetails details ->
        if (details.requested.group == 'org.gradle') {
            details.useVersion '1.4'
        }
    }
}
```

Implement a custom versioning scheme

In some corporate environments, the list of module versions that can be declared in Gradle builds is maintained and audited externally. Dependency resolve rules provide a neat implementation of this pattern:

- In the build script, the developer declares dependencies with the module group and name, but uses a placeholder version, for example: 'default'.
- The 'default' version is resolved to a specific version via a dependency resolve rule, which looks up the version in a corporate catalog of approved modules.

This rule implementation can be neatly encapsulated in a corporate plugin, and shared across all builds within the organisation.

Example 25.49. Using a custom versioning scheme

build.gradle

```
configurations.all {
    resolutionStrategy.eachDependency { DependencyResolveDetails details ->
        if (details.requested.version == 'default') {
            def version = findDefaultVersionInCatalog(details.requested.group, details.requested.name)
            details.useVersion version
        }
    }
}

def findDefaultVersionInCatalog(String group, String name) {
    //some custom logic that resolves the default version into a specific version
    "1.0"
}
```

Blacklisting a particular version with a replacement

Dependency resolve rules provide a mechanism for blacklisting a particular version of a dependency and providing a replacement version. This can be useful if a certain dependency version is broken and should not be used, where a dependency resolve rule causes this version to be replaced with a known good version. One example of a broken module is one that declares a dependency on a library that cannot be found in any of the public repositories, but there are many other reasons why a particular module version is unwanted and a different version is preferred.

In example below, imagine that version 1.2.1 contains important fixes and should always be used in preference to 1.2. The rule provided will enforce just this: any time version 1.2 is encountered it will be replaced with 1.2.1. Note that this is different from a forced version as described above, in that any other versions of this module would not be affected. This means that the 'newest' conflict resolution strategy would still select version 1.3 if this version was also pulled transitively.

Example 25.50. Blacklisting a version with a replacement

build.gradle

```
configurations.all {
    resolutionStrategy.eachDependency { DependencyResolveDetails details ->
        if (details.requested.group == 'org.springframework' && details.requested.name == 'spring-core')
            //prefer different version which contains some necessary fixes
            details.useVersion '1.2.1'
    }
}
```

Substituting a dependency module with a compatible replacement

At times a completely different module can serve as a replacement for a requested module dependency. Examples include using 'groovy' in place of 'groovy-all', or using 'log4j-over-slf4j' instead of 'log4j'. Starting with Gradle 1.5 you can make these substitutions using dependency resolve rules:

Example 25.51. Changing dependency group and/or name at the resolution

build.gradle

```
configurations.all {
    resolutionStrategy.eachDependency { DependencyResolveDetails details ->
        if (details.requested.name == 'groovy-all') {
            //prefer 'groovy' over 'groovy-all':
            details.useTarget group: details.requested.group, name: 'groovy', version: details.requested.version
        }
        if (details.requested.name == 'log4j') {
            //prefer 'log4j-over-slf4j' over 'log4j', with fixed version:
            details.useTarget "org.slf4j:log4j-over-slf4j:1.7.10"
        }
    }
}
```

25.8.4. Dependency Substitution Rules

Dependency substitution rules work similarly to dependency resolve rules. In fact, many capabilities of dependency resolve rules can be implemented with dependency substitution rules. They allow project and module dependencies to be transparently substituted with specified replacements. Unlike dependency resolve rules, dependency substitution rules allow project and module dependencies to be substituted interchangeably.

NOTE: Adding a dependency substitution rule to a configuration changes the timing of when that

configuration is resolved. Instead of being resolved on first use, the configuration is instead resolved when the task graph is being constructed. This can have unexpected consequences if the configuration is being further modified during task execution, or if the configuration relies on modules that are published during execution of another task.

To explain:

- A `Configuration` can be declared as an input to any Task, and that configuration can include project dependencies when it is resolved.
- If a project dependency is an input to a Task (via a configuration), then tasks to build the project artifacts must be added to the task dependencies.
- In order to determine the project dependencies that are inputs to a task, Gradle needs to resolve the `Configuration` inputs.
- Because the Gradle task graph is fixed once task execution has commenced, Gradle needs to perform this resolution prior to executing any tasks.

In the absence of dependency substitution rules, Gradle knows that an external module dependency will never transitively reference a project dependency. This makes it easy to determine the full set of project dependencies for a configuration through simple graph traversal. With this functionality, Gradle can no longer make this assumption, and must perform a full resolve in order to determine the project dependencies.

Substituting an external module dependency with a project dependency

One use case for dependency substitution is to use a locally developed version of a module in place of one that is downloaded from an external repository. This could be useful for testing a local, patched version of a dependency.

The module to be replaced can be declared with or without a version specified.

Example 25.52. Substituting a module with a project

build.gradle

```
configurations.all {
    resolutionStrategy.dependencySubstitution {
        substitute module("org.utils:api") with project(":api")
        substitute module("org.utils:util:2.5") with project(":util")
    }
}
```

Note that a project that is substituted must be included in the multi-project build (via `settings.gradle`). Dependency substitution rules take care of replacing the module dependency with the project dependency and wiring up any task dependencies, but do not implicitly include the project in the build.

Substituting a project dependency with a module replacement

Another way to use substitution rules is to replace a project dependency with a module in a multi-project build. This can be useful to speed up development with a large multi-project build, by allowing a subset of the project dependencies to be downloaded from a repository rather than being built.

The module to be used as a replacement must be declared with a version specified.

Example 25.53. Substituting a project with a module

build.gradle

```
configurations.all {
    resolutionStrategy.dependencySubstitution {
        substitute project(":api") with module("org.utils:api:1.3")
    }
}
```

When a project dependency has been replaced with a module dependency, that project is still included in the overall multi-project build. However, tasks to build the replaced dependency will not be executed in order to build the resolve the depending Configuration.

Conditionally substituting a dependency

A common use case for dependency substitution is to allow more flexible assembly of sub-projects within a multi-project build. This can be useful for developing a local, patched version of an external dependency or for building a subset of the modules within a large multi-project build.

The following example uses a dependency substitution rule to replace any module dependency with the group "org.example", but only if a local project matching the dependency name can be located.

Example 25.54. Conditionally substituting a dependency

build.gradle

```
configurations.all {
    resolutionStrategy.dependencySubstitution.all { DependencySubstitution dependency
        if (dependency.requested instanceof ModuleComponentSelector && dependency.isModule()) {
            def targetProject = findProject(":${dependency.requested.module}")
            if (targetProject != null) {
                dependency.useTarget targetProject
            }
        }
    }
}
```

Note that a project that is substituted must be included in the multi-project build (via settings.gradle). Dependency substitution rules take care of replacing the module dependency with the project dependency, but do not implicitly include the project in the build.

25.8.5. Specifying default dependencies for a configuration

A configuration can be configured with default dependencies to be used if no dependencies are explicitly set for the configuration. A primary use case of this functionality is for developing plugins that make use of versioned tools that the user might override. By specifying default dependencies, the plugin can use a default version of the tool only if the user has not specified a particular version to use.

Example 25.55. Specifying default dependencies on a configuration

build.gradle

```
configurations {
    pluginTool {
        defaultDependencies { dependencies ->
            dependencies.add(this.project.dependencies.create("org.gradle:my-util:1.
        }
    }
}
```

25.8.6. Enabling Ivy dynamic resolve mode

Gradle's Ivy repository implementations support the equivalent to Ivy's dynamic resolve mode. Normally, Gradle will use the `rev` attribute for each dependency definition included in an `ivy.xml` file. In dynamic resolve mode, Gradle will instead prefer the `revConstraint` attribute over the `rev` attribute for a given dependency definition. If the `revConstraint` attribute is not present, the `rev` attribute is used instead.

To enable dynamic resolve mode, you need to set the appropriate option on the repository definition. A couple of examples are shown below. Note that dynamic resolve mode is only available for Gradle's Ivy repositories. It is not available for Maven repositories, or custom Ivy `DependencyResolver` implementations.

Example 25.56. Enabling dynamic resolve mode

build.gradle

```
// Can enable dynamic resolve mode when you define the repository
repositories {
    ivy {
        url "http://repo.mycompany.com/repo"
        resolve.dynamicMode = true
    }
}

// Can use a rule instead to enable (or disable) dynamic resolve mode for all repository
repositories.withType(IvyArtifactRepository) {
    resolve.dynamicMode = true
}
```

25.8.7. Component metadata rules

Each module (also called *component*) has metadata associated with it, such as its group, name, version, dependencies, and so on. This metadata typically originates in the module's descriptor. Metadata rules allow certain parts of a module's metadata to be manipulated from within the build script. They take effect after a module's descriptor has been downloaded, but before it has been selected among all candidate versions. This makes metadata rules another instrument for customizing dependency resolution.

One piece of module metadata that Gradle understands is a module's *status scheme*. This concept, also known from Ivy, models the different levels of maturity that a module transitions through over time. The default status scheme, ordered from least to most mature status, is *integration*, *milestone*, *release*. Apart from a status scheme, a module also has a (current) *status*, which must be one of the values in its status scheme. If not specified in the (Ivy) descriptor, the status defaults to *integration* for Ivy modules and Maven snapshot modules, and *release* for Maven modules that aren't snapshots.

A module's status and status scheme are taken into consideration when a *latest* version selector is resolved. Specifically, *latest.someStatus* will resolve to the highest module version that has status *someSt* or a more mature status. For example, with the default status scheme in place, *latest.integration* will select the highest module version regardless of its status (because *integration* is the least mature status), whereas *latest.release* will select the highest module version with status *release*. Here is what this looks like in code:

Example 25.57. 'Latest' version selector

build.gradle

```
dependencies {
    config1 "org.sample:client:latest.integration"
    config2 "org.sample:client:latest.release"
}

task listConfigs {
    doLast {
        configurations.config1.each { println it.name }
        println()
        configurations.config2.each { println it.name }
    }
}
```

Output of `gradle -q listConfigs`

```
> gradle -q listConfigs
client-1.5.jar

client-1.4.jar
```

The next example demonstrates *latest* selectors based on a custom status scheme declared in a component metadata rule that applies to all modules:

Example 25.58. Custom status scheme

build.gradle

```
dependencies {
    config3 "org.sample:api:latest.silver"
    components {
        all { ComponentMetadataDetails details ->
            if (details.id.group == "org.sample" && details.id.name == "api") {
                details.statusScheme = ["bronze", "silver", "gold", "platinum"]
            }
        }
    }
}
```

Component metadata rules can be applied to a specified module. Modules must be specified in the form of "group:module".

Example 25.59. Custom status scheme by module

build.gradle

```
dependencies {
    config4 "org.sample:lib:latest.prod"
    components {
        withModule('org.sample:lib') { ComponentMetadataDetails details ->
            details.statusScheme = ["int", "rc", "prod"]
        }
    }
}
```

Gradle can also create component metadata rules utilizing Ivy-specific metadata for modules resolved from an Ivy repository. Values from the Ivy descriptor are made available via the `IvyModuleDescriptor` interface.

Example 25.60. Ivy component metadata rule

build.gradle

```
dependencies {
    config6 "org.sample:lib:latest.rc"
    components {
        withModule("org.sample:lib") { ComponentMetadataDetails details, IvyModuleDescriptor descriptor ->
            if (descriptor.branch == 'testing') {
                details.status = "rc"
            }
        }
    }
}
```

Note that any rule that declares specific arguments must *always* include a `ComponentMetadataDetails` argument as the first argument. The second Ivy metadata argument is optional.

Component metadata rules can also be defined using a *rule source* object. A rule source object is any object

that contains exactly one method that defines the rule action and is annotated with `@Mutate`.

This method:

- must return `void`.
- must have `ComponentMetadataDetails` as the first argument.
- may have an additional parameter of type `IvyModuleDescriptor`.

Example 25.61. Rule source component metadata rule

build.gradle

```
dependencies {
    config5 "org.sample:api:latest.gold"
    components {
        withModule('org.sample:api', new CustomStatusRule())
    }
}

class CustomStatusRule {
    @Mutate
    void setStatusScheme(ComponentMetadataDetails details) {
        details.statusScheme = ["bronze", "silver", "gold", "platinum"]
    }
}
```

25.8.8. Component Selection Rules

Component selection rules may influence which component instance should be selected when multiple versions are available that match a version selector. Rules are applied against every available version and allow the version to be explicitly rejected by rule. This allows Gradle to ignore any component instance that does not satisfy conditions set by the rule. Examples include:

- For a dynamic version like '1.+' certain versions may be explicitly rejected from selection
- For a static version like '1.4' an instance may be rejected based on extra component metadata such as the Ivy branch attribute, allowing an instance from a subsequent repository to be used.

Rules are configured via the `ComponentSelectionRules` object. Each rule configured will be called with a `ComponentSelection` object as an argument which contains information about the candidate version being considered. Calling `ComponentSelection.reject(java.lang.String)` causes the given candidate version to be explicitly rejected, in which case the candidate will not be considered for the selector.

The following example shows a rule that disallows a particular version of a module but allows the dynamic version to choose the next best candidate.

Example 25.62. Component selection rule

build.gradle

```
configurations {
    rejectConfig {
        resolutionStrategy {
            componentSelection {
                // Accept the highest version matching the requested version that is
                all { ComponentSelection selection ->
                    if (selection.candidate.group == 'org.sample' && selection.candidate.version == "1.5") {
                        selection.reject("version 1.5 is broken for 'org.sample:api'")
                    }
                }
            }
        }
    }
}

dependencies {
    rejectConfig "org.sample:api:1.+"
}
```

Note that version selection is applied starting with the highest version first. The version selected will be the first version found that all component selection rules accept. A version is considered accepted no rule explicitly rejects it.

Similarly, rules can be targeted at specific modules. Modules must be specified in the form of "group:module".

Example 25.63. Component selection rule with module target

build.gradle

```
configurations {
    targetConfig {
        resolutionStrategy {
            componentSelection {
                withModule("org.sample:api") { ComponentSelection selection ->
                    if (selection.candidate.version == "1.5") {
                        selection.reject("version 1.5 is broken for 'org.sample:api'")
                    }
                }
            }
        }
    }
}
```

Component selection rules can also consider component metadata when selecting a version. Possible metadata arguments that can be considered are `ComponentMetadata` and `IvyModuleDescriptor`.

Example 25.64. Component selection rule with metadata

build.gradle

```
configurations {
    metadataRulesConfig {
        resolutionStrategy {
            componentSelection {
                // Reject any versions with a status of 'experimental'
                all { ComponentSelection selection, ComponentMetadata metadata, IvyModuleDescriptor descriptor }
                if (selection.candidate.group == 'org.sample' && metadata.status == 'experimental') {
                    selection.reject("don't use experimental candidates from 'org.sample'")
                }
            }
            // Accept the highest version with either a "release" branch or a "staging" branch
            withModule('org.sample:api') { ComponentSelection selection, IvyModuleDescriptor descriptor }
            if (descriptor.branch != "release" && metadata.status != 'milestone') {
                selection.reject("'org.sample:api' must have testing branch")
            }
        }
    }
}
```

Note that a `ComponentSelection` argument is *always* required as the first parameter when declaring a component selection rule with additional Ivy metadata parameters, but the metadata parameters can be declared in any order.

Lastly, component selection rules can also be defined using a *rule source* object. A rule source object is any object that contains exactly one method that defines the rule action and is annotated with `@Mutate`.

This method:

- must return void.
- must have `ComponentSelection` as the first argument.
- may have additional parameters of type `ComponentMetadata` and/or `IvyModuleDescriptor`.

Example 25.65. Component selection rule using a rule source object

build.gradle

```
class RejectTestBranch {
    @Mutate
    void evaluateRule(ComponentSelection selection, IvyModuleDescriptor ivy) {
        if (ivy.branch == "test") {
            selection.reject("reject test branch")
        }
    }
}

configurations {
    ruleSourceConfig {
        resolutionStrategy {
            componentSelection {
                all new RejectTestBranch()
            }
        }
    }
}
```

25.8.9. Module replacement rules

Module replacement rules allow a build to declare that a legacy library has been replaced by a new one. A good example when a new library replaced a legacy one is the "google-collections" -> "guava" migration. The team that created google-collections decided to change the module name from "com.google.collections:google-collections" into "com.google.guava:guava". This is a legal scenario in the industry: teams need to be able to change the names of products they maintain, including the module coordinates. Renaming of the module coordinates has impact on conflict resolution.

To explain the impact on conflict resolution, let's consider the "google-collections" -> "guava" scenario. It may happen that both libraries are pulled into the same dependency graph. For example, "our" project depends on guava but some of our dependencies pull in a legacy version of google-collections. This can cause runtime errors, for example during test or application execution. Gradle does not automatically resolve the google-collections VS guava conflict because it is not considered as a "version conflict". It's because the module coordinates for both libraries are completely different and conflict resolution is activated when "group" and "name" coordinates are the same but there are different versions available in the dependency graph (for more info, please refer to the section on conflict resolution). Traditional remedies to this problem are:

- Declare exclusion rule to avoid pulling in "google-collections" to graph. It is probably the most popular approach.
- Avoid dependencies that pull in legacy libraries.
- Upgrade the dependency version if the new version no longer pulls in a legacy library.
- Downgrade to "google-collections". It's not recommended, just mentioned for completeness.

Traditional approaches work but they are not general enough. For example, an organisation wants to resolve the google-collections VS guava conflict resolution problem in all projects. Starting from Gradle 2.2 it is

possible to declare that certain module was replaced by other. This enables organisations to include the information about module replacement in the corporate plugin suite and resolve the problem holistically for all Gradle-powered projects in the enterprise.

Example 25.66. Declaring module replacement

build.gradle

```
dependencies {
    modules {
        module( "com.google.collections:google-collections" ) {
            replacedBy( "com.google.guava:guava" )
        }
    }
}
```

For more examples and detailed API, please refer to the DSL reference for `ComponentMetadataHandler`.

What happens when we declare that "google-collections" are replaced by "guava"? Gradle can use this information for conflict resolution. Gradle will consider every version of "guava" newer/better than any version of "google-collections". Also, Gradle will ensure that only guava jar is present in the classpath / resolved file list. Please note that if only "google-collections" appears in the dependency graph (e.g. no "guava") Gradle will not eagerly replace it with "guava". Module replacement is an information that Gradle uses for resolving conflicts. If there is no conflict (e.g. only "google-collections" or only "guava" in the graph) the replacement information is not used.

Currently it is not possible to declare that certain modules is replaced by a set of modules. However, it is possible to declare that multiple modules are replaced by a single module.

25.9. The dependency cache

Gradle contains a highly sophisticated dependency caching mechanism, which seeks to minimise the number of remote requests made in dependency resolution, while striving to guarantee that the results of dependency resolution are correct and reproducible.

The Gradle dependency cache consists of 2 key types of storage:

- A file-based store of downloaded artifacts, including binaries like jars as well as raw downloaded meta-data like POM files and Ivy files. The storage path for a downloaded artifact includes the SHA1 checksum, meaning that 2 artifacts with the same name but different content can easily be cached.
- A binary store of resolved module meta-data, including the results of resolving dynamic versions, module descriptors, and artifacts.

Separating the storage of downloaded artifacts from the cache metadata permits us to do some very powerful things with our cache that would be difficult with a transparent, file-only cache layout.

The Gradle cache does not allow the local cache to hide problems and create other mysterious and difficult to debug behavior that has been a challenge with many build tools. This new behavior is implemented in a bandwidth and storage efficient way. In doing so, Gradle enables reliable and reproducible enterprise builds.

25.9.1. Key features of the Gradle dependency cache

Separate metadata cache

Gradle keeps a record of various aspects of dependency resolution in binary format in the metadata cache. The information stored in the metadata cache includes:

- The result of resolving a dynamic version (e.g. 1 . +) to a concrete version (e.g. 1 . 2).
- The resolved module metadata for a particular module, including module artifacts and module dependencies.
- The resolved artifact metadata for a particular artifact, including a pointer to the downloaded artifact file.
- The *absence* of a particular module or artifact in a particular repository, eliminating repeated attempts to access a resource that does not exist.

Every entry in the metadata cache includes a record of the repository that provided the information as well as a timestamp that can be used for cache expiry.

Repository caches are independent

As described above, for each repository there is a separate metadata cache. A repository is identified by its URL, type and layout. If a module or artifact has not been previously resolved from *this repository*, Gradle will attempt to resolve the module against the repository. This will always involve a remote lookup on the repository, however in many cases no download will be required (see the section called “Artifact reuse”, below).

Dependency resolution will fail if the required artifacts are not available in any repository specified by the build, even if the local cache has a copy of this artifact which was retrieved from a different repository. Repository independence allows builds to be isolated from each other in an advanced way that no build tool has done before. This is a key feature to create builds that are reliable and reproducible in any environment.

Artifact reuse

Before downloading an artifact, Gradle tries to determine the checksum of the required artifact by downloading the sha file associated with that artifact. If the checksum can be retrieved, an artifact is not downloaded if an artifact already exists with the same id and checksum. If the checksum cannot be retrieved from the remote server, the artifact will be downloaded (and ignored if it matches an existing artifact).

As well as considering artifacts downloaded from a different repository, Gradle will also attempt to reuse artifacts found in the local Maven Repository. If a candidate artifact has been downloaded by Maven, Gradle will use this artifact if it can be verified to match the checksum declared by the remote server.

Checksum based storage

It is possible for different repositories to provide a different binary artifact in response to the same artifact identifier. This is often the case with Maven SNAPSHOT artifacts, but can also be true for any artifact which is republished without changing its identifier. By caching artifacts based on their SHA1 checksum, Gradle is able to maintain multiple versions of the same artifact. This means that when resolving against one repository Gradle will never overwrite the cached artifact file from a different repository. This is done without requiring a separate artifact file store per repository.

Cache Locking

The Gradle dependency cache uses file-based locking to ensure that it can safely be used by multiple Gradle processes concurrently. The lock is held whenever the binary meta-data store is being read or written, but is released for slow operations such as downloading remote artifacts.

25.9.2. Command line options to override caching

Offline

The `--offline` command line switch tells Gradle to always use dependency modules from the cache, regardless if they are due to be checked again. When running with offline, Gradle will never attempt to access the network to perform dependency resolution. If required modules are not present in the dependency cache, build execution will fail.

Refresh

At times, the Gradle Dependency Cache can be out of sync with the actual state of the configured repositories. Perhaps a repository was initially misconfigured, or perhaps a “non-changing” module was published incorrectly. To refresh all dependencies in the dependency cache, use the `--refresh-dependencies` option on the command line.

The `--refresh-dependencies` option tells Gradle to ignore all cached entries for resolved modules and artifacts. A fresh resolve will be performed against all configured repositories, with dynamic versions recalculated, modules refreshed, and artifacts downloaded. However, where possible Gradle will check if the previously downloaded artifacts are valid before downloading again. This is done by comparing published SHA1 values in the repository with the SHA1 values for existing downloaded artifacts.

25.9.3. Fine-tuned control over dependency caching

You can fine-tune certain aspects of caching using the `ResolutionStrategy` for a configuration.

By default, Gradle caches dynamic versions for 24 hours. To change how long Gradle will cache the resolved version for a dynamic version, use:

Example 25.67. Dynamic version cache control

build.gradle

```
configurations.all {  
    resolutionStrategy.cacheDynamicVersionsFor 10, 'minutes'  
}
```

By default, Gradle caches changing modules for 24 hours. To change how long Gradle will cache the meta-data and artifacts for a changing module, use:

Example 25.68. Changing module cache control

build.gradle

```
configurations.all {  
    resolutionStrategy.cacheChangingModulesFor 4, 'hours'  
}
```

For more details, take a look at the API documentation for `ResolutionStrategy`.

25.10. Strategies for transitive dependency management

Many projects rely on the Maven Central repository. This is not without problems.

- The Maven Central repository can be down or can be slow to respond.
- The POM files of many popular projects specify dependencies or other configuration that are just plain wrong (for instance, the POM file of the “commons-httpclient-3.0” module 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 Maven Central repository you are likely to need an additional custom repository, because:

- You might need dependencies that are not uploaded to Maven Central yet.
- You want to deal properly with invalid metadata in a Maven Central POM file.
- You don't want to expose people to the downtimes or slow response of Maven Central, if they just want to build your project.

It is not a big deal to set-up a custom repository, ^[14] but it can be tedious to keep it up to date. For a new version, you always have to create the new XML descriptor and the directories. 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, not to mention a backup of these. 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 managers like Artifactory or Nexus make this easier, but most open source projects don't usually have a host for those products. This is changing with new services like Bintray that let developers host and distribute their release binaries using a self-service repository platform. Bintray also supports sharing approved artifacts through the JCenter public repository to provide a single resolution address for all popular OSS Java artifacts (see Section 25.6.2, “Maven JCenter repository”).

This is a common reason why many 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 your source code control system 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 file and maintaining custom XML descriptors, then *Client Modules* offer an alternative. However, you can still use a Maven2 repo or your custom repository as a repository for *jars only* and still enjoy *transitive* dependency management. Or you can only provide client modules for POMs with bad metadata. For the jars and the correct POMs you still use the remote repository.

25.10.1. 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. This will directly express your first level dependencies and your transitive dependencies (see Section 25.4.8, “Optional attributes”). The problem with this is that Gradle dependency management will see this as specifying all dependencies as first level dependencies. The dependency reports won'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.

[11] Gradle supports partial multiproject builds (see Chapter 26, *Multi-project Builds*).

[12] <http://books.sonatype.com/mvnref-book/reference/pom-relationships-sect-project-relationships.html>

[13] At <http://ant.apache.org/ivy/history/latest-milestone/concept.html> you can learn more about ivy patterns.

[14] If you want to shield your project from the downtimes of Maven Central 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.

26

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.

A multi-project build in gradle consists of one root project, and one or more subprojects that may also have subprojects.

26.1. Cross project configuration

While each subproject could configure itself in complete isolation of the other subprojects, it is common that subprojects share common traits. It is then usually preferable to share configurations among projects, so the same configuration affects several subprojects.

Let's start with a very simple multi-project build. 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.

26.1.1. Configuration and execution

Section 22.1, “Build phases” describes the phases of every Gradle build. Let's zoom into the configuration and execution phases of a multi-project build. Configuration here means executing the `build.gradle` file of a project, which implies e.g. downloading all plugins that were declared using `'apply plugin'`. By default, the configuration of all projects happens before any task is executed. This means that when a single task, from a single project is requested, *all* projects of multi-project build are configured first. The reason every project needs to be configured is to support the flexibility of accessing and changing any part of the Gradle project model.

Configuration on demand

The *Configuration injection* feature and access to the complete project model are possible because every project is configured before the execution phase. Yet, this approach may not be the most efficient in a very large multi-project build. There are Gradle builds with a hierarchy of hundreds of subprojects. The configuration time of huge multi-project builds may become noticeable. Scalability is an important requirement for Gradle. Hence, starting from version 1.4 a new incubating 'configuration on demand' mode is introduced.

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

incubating so not every build is guaranteed to work correctly. The feature should work very well for multi-project builds that have decoupled projects (Section 26.9, “Decoupled Projects”). In “configuration on demand” mode, projects are configured as follows:

- The root project is always configured. This way the typical common configuration is supported (allprojects or subprojects script blocks).
- The project in the directory where the build is executed is also configured, but only when Gradle is executed without any tasks. This way the default tasks behave correctly when projects are configured on demand.
- The standard project dependencies are supported and makes relevant projects configured. If project A has a compile dependency on project B then building A causes configuration of both projects.
- The task dependencies declared via task path are supported and cause relevant projects to be configured. Example: `someTask.dependsOn(":someOtherProject:someOtherTask")`
- A task requested via task path from the command line (or Tooling API) causes the relevant project to be configured. For example, building `'projectA:projectB:someTask'` causes configuration of projectB.

Eager to try out this new feature? To configure on demand with every build run see Section 12.1, “Configuring the build environment via `gradle.properties`”. To configure on demand just for a given build please see Appendix D, *Gradle Command Line*.

26.1.2. Defining common behavior

Let's look at some examples with the following project tree. This is a multi-project build with a root project named `water` and a subproject named `bluewhale`.

Example 26.1. 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` in the ‘-all’ distribution of Gradle.

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 26.2. Build script of water (parent) project

build.gradle

```
Closure cl = { task -> println "I'm $task.project.name" }
task('hello').doLast(cl)
project(':bluewhale') {
    task('hello').doLast(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 26.3. 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/v` in the ‘-all’ distribution of Gradle.

settings.gradle

```
include 'bluewhale', 'krill'
```

Now we rewrite the `water` build script and boil it down to a single line.

Example 26.4. Water project build script

build.gradle

```
allprojects {  
    task hello {  
        doLast { 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.

Another possibility for sharing configuration is to use a common external script. See Section 43.3, “Configuring the project using an external build script” for more information.

26.2. Subproject configuration

The Project API also provides a property for accessing the subprojects only.

26.2.1. Defining common behavior

Example 26.5. Defining common behavior of all projects and subprojects

build.gradle

```
allprojects {
    task hello {
        doLast { task ->
            println "I'm $task.project.name"
        }
    }
}
subprojects {
    hello {
        doLast {
            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
```

You may notice that there are two code snippets referencing the “hello” task. The first one, which uses the “task” keyword, constructs the task and provides its base configuration. The second piece doesn't use the “task” keyword, as it is further configuring the existing “hello” task. You may only construct a task once in a project, but you may add any number of code blocks providing additional configuration.

26.2.2. 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 26.6. Defining specific behaviour for particular project

build.gradle

```
allprojects {
    task hello {
        doLast { task ->
            println "I'm $task.project.name"
        }
    }
}
subprojects {
    hello {
        doLast {
            println "- I depend on water"
        }
    }
}
project(':bluewhale').hello {
    doLast {
        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 26.7. 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` in the ‘-all’ distribution of Gradle.

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 {  
        doLast { task ->  
            println "I'm $task.project.name"  
        }  
    }  
}  
subprojects {  
    hello {  
        doLast {  
            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.
```


26.2.3. 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.

Filtering by name

Example 26.8. 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/addTropicalFish](#) in the ‘-all’ distribution of Gradle.

settings.gradle

```
include 'bluewhale', 'krill', 'tropicalFish'
```

build.gradle

```
allprojects {  
    task hello {  
        doLast { task ->  
            println "I'm $task.project.name"  
        }  
    }  
}  
subprojects {  
    hello {  
        doLast {  
            println "- I depend on water"  
        }  
    }  
}  
configure(subprojects.findAll {it.name != 'tropicalFish'}) {  
    hello {  
        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 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.

Filtering by properties

Using the project name for filtering is one option. Using extra project properties is another. (See Section 18.4.2, “Extra properties” for more information on extra properties.)

Example 26.9. 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/multiplatform/tropicalFish` in the ‘-all’ distribution of Gradle.

settings.gradle

```
include 'bluewhale', 'krill', 'tropicalFish'
```

bluewhale/build.gradle

```
ext.arctic = true  
hello.doLast {  
    println "- I'm the largest animal that has ever lived on this planet."  
}
```

krill/build.gradle

```
ext.arctic = true  
hello.doLast {  
    println "- The weight of my species in summer is twice as heavy as all human beings"  
}
```

tropicalFish/build.gradle

```
ext.arctic = false
```

build.gradle

```

allprojects {
    task hello {
        doLast { 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 26.6, “Dependencies - Which dependencies?”

26.3. Execution rules for multi-project builds

When we executed the `hello` task from the root project dir, things behaved in an intuitive way. All the `hello` 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 26.10. 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 26.11. Evaluation and execution of projects

`bluewhale/build.gradle`

```
ext.arctic = true
hello {
    doLast {
        println "- I'm the largest animal that has ever lived on this planet."
    }
}

task distanceToIceberg {
    doLast {
        println '20 nautical miles'
    }
}
```

`krill/build.gradle`

```
ext.arctic = true
hello {
    doLast {
        println "- The weight of my species in summer is twice as heavy as all human"
    }
}

task distanceToIceberg {
    doLast {
        println '5 nautical miles'
    }
}
```

Output of `gradle -q distanceToIceberg`

```
> gradle -q distanceToIceberg
20 nautical miles
5 nautical miles
```

Here's the output without the `-q` option:

Example 26.12. 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!

26.4. 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 26.5, “Project and task paths”):

Example 26.13. 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 using the name matching mechanism described above.

26.5. Project and task paths

A project path has the following pattern: It starts with an optional colon, which denotes the root project. The root project is the only project in a path that is not specified by its name. The rest of a project path is a colon-separated sequence of project names, where the next project is a subproject of the previous project.

The path of a task is simply its project path plus the task name, like “`: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.

26.6. Dependencies - Which dependencies?

The examples from the last section were special, as the projects had no *Execution Dependencies*. They had only *Configuration Dependencies*. The following sections illustrate the differences between these two types of dependencies.

26.6.1. Execution dependencies

Dependencies and execution order

Example 26.14. Dependencies and execution order

Build layout

```
messages/  
  build.gradle  
  settings.gradle  
consumer/  
  build.gradle  
producer/  
  build.gradle
```

Note: The code for this example can be found at [samples/userguide/multiplatform/dependencies](#) in the ‘-all’ distribution of Gradle.

build.gradle

```
ext.producerMessage = null
```

settings.gradle

```
include 'consumer', 'producer'
```

consumer/build.gradle

```
task action {  
  doLast {  
    println("Consuming message: ${rootProject.producerMessage}")  
  }  
}
```

producer/build.gradle

```
task action {  
  doLast {  
    println "Producing message:"  
    rootProject.producerMessage = 'Watch the order of execution.'  
  }  
}
```

Output of **gradle -q action**

```
> gradle -q action  
Consuming message: null  
Producing message:
```

This didn't quite do what we want. If nothing else is defined, Gradle executes the task in alphanumeric order. Therefore, Gradle will execute “:consumer:action” before “:producer:action”. Let's try to solve this with a hack and rename the producer project to “aProducer”.

Example 26.15. Dependencies and execution order

Build layout

```
messages/  
  build.gradle  
  settings.gradle  
  aProducer/  
    build.gradle  
  consumer/  
    build.gradle
```

build.gradle

```
ext.producerMessage = null
```

settings.gradle

```
include 'consumer', 'aProducer'
```

aProducer/build.gradle

```
task action {  
  doLast {  
    println "Producing message:"  
    rootProject.producerMessage = 'Watch the order of execution.'  
  }  
}
```

consumer/build.gradle

```
task action {  
  doLast {  
    println("Consuming message: ${rootProject.producerMessage}")  
  }  
}
```

Output of **gradle -q action**

```
> gradle -q action  
Producing message:  
Consuming message: Watch the order of execution.
```

We can show where this hack doesn't work if we now switch to the consumer dir and execute the build.

Example 26.16. Dependencies and execution order

Output of **gradle -q action**

```
> gradle -q action  
Consuming message: null
```

The problem is that the two “action” tasks are unrelated. 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” task was down the hierarchy and therefore it was the only task that was executed. We need something better than this hack.

Declaring dependencies

Example 26.17. Declaring dependencies

Build layout

```
messages/  
  build.gradle  
  settings.gradle  
  consumer/  
    build.gradle  
  producer/  
    build.gradle
```

Note: The code for this example can be found at `samples/userguide/multiplatform/dependencies` in the ‘-all’ distribution of Gradle.

build.gradle

```
ext.producerMessage = null
```

settings.gradle

```
include 'consumer', 'producer'
```

consumer/build.gradle

```
task action(dependsOn: ":producer:action") {  
    doLast {  
        println("Consuming message: ${rootProject.producerMessage}")  
    }  
}
```

producer/build.gradle

```
task action {  
    doLast {  
        println "Producing message:"  
        rootProject.producerMessage = '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 26.18. Declaring dependencies

Output of `gradle -q action`

```
> gradle -q action  
Producing message:  
Consuming message: Watch the order of execution.
```

This is now working better because we have declared that the “action” task in the “consumer” project has an *execution dependency* on the “action” task in the “producer” project.

The nature of cross project task dependencies

Of course, task dependencies across different projects are not limited to tasks with the same name. Let's change the naming of our tasks and execute the build.

Example 26.19. Cross project task dependencies

consumer/build.gradle

```
task consume(dependsOn: ':producer:produce') {
    doLast {
        println("Consuming message: ${rootProject.producerMessage}")
    }
}
```

producer/build.gradle

```
task produce {
    doLast {
        println "Producing message:"
        rootProject.producerMessage = 'Watch the order of execution.'
    }
}
```

Output of `gradle -q consume`

```
> gradle -q consume
Producing message:
Consuming message: Watch the order of execution.
```

26.6.2. Configuration time dependencies

Let's see one more example with our producer-consumer build before we enter *Java* land. We add a property to the “producer” project and create a configuration time dependency from “consumer” to “producer”.

Example 26.20. Configuration time dependencies

consumer/build.gradle

```
def message = rootProject.producerMessage

task consume {
    doLast {
        println("Consuming message: " + message)
    }
}
```

producer/build.gradle

```
rootProject.producerMessage = 'Watch the order of evaluation.'
```

Output of **gradle -q consume**

```
> gradle -q consume
Consuming message: null
```

The default *evaluation* order of projects is alphanumeric (for the same nesting level). Therefore the “consumer” project is evaluated before the “producer” project and the “producerMessage” value is set *after* it is read by the “consumer” project. Gradle offers a solution for this.

Example 26.21. Configuration time dependencies - evaluationDependsOn

consumer/build.gradle

```
evaluationDependsOn(':producer')

def message = rootProject.producerMessage

task consume {
    doLast {
        println("Consuming message: " + message)
    }
}
```

Output of **gradle -q consume**

```
> gradle -q consume
Consuming message: Watch the order of evaluation.
```

The use of the “evaluationDependsOn” command results in the evaluation of the “producer” project *before* the “consumer” project is evaluated. This example is a bit contrived to show the mechanism. In *this* case there would be an easier solution by reading the key property at execution time.

Example 26.22. Configuration time dependencies

consumer/build.gradle

```
task consume {
    doLast {
        println("Consuming message: ${rootProject.producerMessage}")
    }
}
```

Output of **gradle -q consume**

```
> gradle -q consume
Consuming message: Watch the order of evaluation.
```

Configuration dependencies are very different from execution dependencies. Configuration dependencies are between projects whereas execution dependencies are always resolved to task dependencies. Also note that all projects are always configured, even when you start the build from a subproject. The default configuration order is top down, which is usually what is needed.

To change the default configuration order to “bottom up”, use the “`evaluationDependsOnChildren()`” method instead.

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 also create a *configuration* dependency between the two projects. Therefore it is likely that you don't have to define configuration dependencies explicitly.

26.6.3. Real life examples

Gradle's multi-project features are driven by real life use cases. One good example consists of two web application projects and a parent project that creates a distribution including the two web applications. ^[15] For the example we use only one build script and do *cross project configuration*.

Example 26.23. 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` in the ‘-all’ distribution of Gradle.

settings.gradle

```
include 'date', 'hello'
```

build.gradle

```
allprojects {
    apply plugin: 'java'
    group = 'org.gradle.sample'
    version = '1.0'
}

subprojects {
    apply plugin: 'war'
    repositories {
        mavenCentral()
    }
    dependencies {
        compile "javax.servlet:servlet-api:2.5"
    }
}

task explodedDist(type: Copy) {
    into "$buildDir/explodedDist"
    subprojects {
        from tasks.withType(War)
    }
}
```

We have an interesting set of dependencies. Obviously the `date` and `hello` projects 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 dependency patterns are daily bread in the problem space of multi-project builds. If a build system

does not support these patterns, you either can't solve your problem or you need to do ugly hacks which are hard to maintain and massively impair your productivity as a build master.

26.7. Project lib dependencies

What if one project 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 25.4.3, “Project dependencies”, Gradle offers project lib dependencies for this.

Example 26.24. Project lib dependencies

Build layout

```
java/
  settings.gradle
  build.gradle
  api/
    src/main/java/
      org/gradle/sample/
        api/
          Person.java
        apiImpl/
          PersonImpl.java
  services/personService/
    src/
      main/java/
        org/gradle/sample/services/
          PersonService.java
      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](#) in the ‘-all’ distribution of Gradle.

We have the projects “shared”, “api” and “personService”. The “personService” project has a lib dependency on the other two projects. The “api” project has a lib dependency on the “shared” project.^[16]

Example 26.25. Project lib dependencies

settings.gradle

```
include 'api', 'shared', 'services:personService'
```

build.gradle

```
subprojects {
    apply plugin: 'java'
    group = 'org.gradle.sample'
    version = '1.0'
    repositories {
        mavenCentral()
    }
    dependencies {
        testCompile "junit:junit:4.12"
    }
}

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

project(':services:personService') {
    dependencies {
        compile project(':shared'), project(':api')
    }
}
```

All the build logic is in the “build.gradle” file of the root project. ^[17] A “*lib*” dependency is a special form of an execution dependency. It causes the other project to be built first and adds the jar with the classes of the other project to the classpath. It also adds the dependencies of the other project to the classpath. So you can enter the “api” directory and trigger a “**gradle compile**”. First the “shared” project is built and then the “api” project is built. 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 26.26. Fine grained control over dependencies

build.gradle

```
subprojects {
    apply plugin: 'java'
    group = 'org.gradle.sample'
    version = '1.0'
}

project(':api') {
    configurations {
        spi
    }
    dependencies {
        compile project(':shared')
    }
    task spiJar(type: Jar) {
        baseName = 'api-spi'
        dependsOn classes
        from sourceSets.main.output
        include('org/gradle/sample/api/**')
    }
    artifacts {
        spi spiJar
    }
}

project(':services:personService') {
    dependencies {
        compile project(':shared')
        compile project(path: ':api', configuration: 'spi')
        testCompile "junit:junit:4.12", 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”.

26.7.1. Disabling 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.

26.8. Parallel project execution

With more and more CPU cores available on developer desktops and CI servers, it is important that Gradle is able to fully utilise these processing resources. More specifically, the parallel execution attempts to:

- Reduce total build time for a multi-project build where execution is IO bound or otherwise does not consume all available CPU resources.
- Provide faster feedback for execution of small projects without awaiting completion of other projects.

Although Gradle already offers parallel test execution via `Test.setMaxParallelForks(int)` the feature described in this section is parallel execution at a project level. Parallel execution is an incubating feature. Please use it and let us know how it works for you.

Parallel project execution allows the separate projects in a decoupled multi-project build to be executed in parallel (see also: Section 26.9, “Decoupled Projects”). While parallel execution does not strictly require decoupling at configuration time, the long-term goal is to provide a powerful set of features that will be available for fully decoupled projects. Such features include:

- the section called “Configuration on demand”.
- Configuration of projects in parallel.
- Re-use of configuration for unchanged projects.
- Project-level up-to-date checks.
- Using pre-built artifacts in the place of building dependent projects.

How does parallel execution work? First, you need to tell Gradle to use the parallel mode. You can use the command line argument (Appendix D, *Gradle Command Line*) or configure your build environment (Section 12.1, “Configuring the build environment via `gradle.properties`”). Unless you provide a specific number of parallel threads Gradle attempts to choose the right number based on available CPU cores. Every parallel worker exclusively owns a given project while executing a task. This means that 2 tasks from the same project are never executed in parallel. Therefore only multi-project builds can take advantage of parallel execution. Task dependencies are fully supported and parallel workers will start executing upstream tasks first. Bear in mind that the alphabetical scheduling of decoupled tasks, known from the sequential execution, does not really work in parallel mode. You need to make sure the task dependencies are declared correctly to avoid ordering issues.

Warning: Be aware that task ordering is not strictly enforced when using parallel execution and can lead to unexpected results. A common case that surfaces this limitation is the use of the `clean` task provided by the `base` plugin in combination with any other task producing an output for a multi-project build if executed in parallel. For example let us assume a multi-project build with project A and project B where B depends on A. Running `gradle clean build --parallel` could lead to the following situation:

- `A:clean` is executed after `A:jar`.
- B depends on A and needs the JAR file of A. However, `B:classes` fails as it was executed after `A:clean` which deleted the JAR file B depends on for compilation.

Furthermore, the tasks `clean` and `classes` could run at the same time and delete files that are needed for compilation across project boundaries. Gradle emits a warning for those situations. Future versions of Gradle will provide an appropriate fix.

26.9. Decoupled Projects

Gradle allows any project to access any other project during both the configuration and execution phases. While this provides a great deal of power and flexibility to the build author, it also limits the flexibility that Gradle has when building those projects. For instance, this effectively prevents Gradle from correctly building multiple projects in parallel, configuring only a subset of projects, or from substituting a pre-built artifact in place of a project dependency.

Two projects are said to be *decoupled* if they do not directly access each other's project model. Decoupled projects may only interact in terms of declared dependencies: project dependencies (Section 25.4.3, “Project dependencies”) and/or task dependencies (Section 16.5, “Task dependencies”). Any other form of project interaction (i.e. by modifying another project object or by reading a value from another project object) causes the projects to be coupled. The consequence of coupling during the configuration phase is that if gradle is invoked with the 'configuration on demand' option, the result of the build can be flawed in several ways. The consequence of coupling during execution phase is that if gradle is invoked with the parallel option, one project task runs too late to influence a task of a project building in parallel. Gradle does not attempt to detect coupling and warn the user, as there are too many possibilities to introduce coupling.

A very common way for projects to be coupled is by using configuration injection (Section 26.1, “Cross project configuration”). It may not be immediately apparent, but using key Gradle features like the `allprojects` and `subprojects` keywords automatically cause your projects to be coupled. This is because these keywords are used in a `build.gradle` file, which defines a project. Often this is a “root project” that does nothing more than define common configuration, but as far as Gradle is concerned this root project is still a fully-fledged project, and by using `allprojects` that project is effectively coupled to all other projects. Coupling of the root project to subprojects does not impact 'configuration on demand', but using the `allprojects` and `subprojects` in any subproject's `build.gradle` file will have an impact.

This means that using any form of shared build script logic or configuration injection (`allprojects`, `subprojects`, etc.) will cause your projects to be coupled. As we extend the concept of project decoupling and provide features that take advantage of decoupled projects, we will also introduce new features to help you to solve common use cases (like configuration injection) without causing your projects to be coupled.

In order to make good use of cross project configuration without running into issues for parallel and 'configuration on demand' options, follow these recommendations:

- Avoid a subproject's `build.gradle` referencing other subprojects; preferring cross configuration from the root project.
- Avoid changing the configuration of other projects at execution time.

26.10. 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.

Look at Example 26.25, “Project lib dependencies”. In this example, the “`:services:personservice`” project depends on both the “`:api`” and “`:shared`” projects. The “`:api`” project also depends on the “`:shared`” project.

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 26.27. Build and Test Single Project

Output of **gradle :api:build**

```
> gradle :api:build
:shared:compileJava
:shared:processResources
:shared:classes
:shared:jar
: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 the “:api” project (knowing that you are only changing files in this one project), you may not want to even suffer the expense of building “:shared:compile” 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 26.28. 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 26.29. Build and Test Depended On Projects

Output of **gradle :api:buildNeeded**

```
> gradle :api:buildNeeded
:shared:compileJava
:shared:processResources
:shared:classes
:shared:jar
: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
:shared:buildNeeded
:api:buildNeeded

BUILD SUCCESSFUL

Total time: 1 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 26.30. Build and Test Dependent Projects

Output of **gradle :api:buildDependents**

```
> gradle :api:buildDependents
:shared:compileJava
:shared:processResources
:shared:classes
:shared:jar
:api:compileJava
:api:processResources
:api:classes
:api:jar
:api:assemble
:api:compileTestJava
:api:processTestResources
:api:testClasses
:api:test
:api:check
:api:build
: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
:services:personService:buildDependents
:api:buildDependents

BUILD SUCCESSFUL

Total time: 1 secs
```

Finally, you may want to build and test everything in all projects. Any task you run in the root project folder 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.

26.11. Multi Project and buildSrc

Section 43.4, “Build sources in the `buildSrc` project” tells us that we can place build logic to be compiled and tested in the special `buildSrc` directory. In a multi project build, there can only be one `buildSrc` directory which must be located in the root directory.

26.12. 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.

26.13. 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 five elements you need to remember: `allprojects`, `subprojects`, `evaluationDependsOn`, `evaluationDependsOnC` and project lib dependencies. ^[18] With those elements, and keeping in mind that Gradle has a distinct configuration and execution phase, you already have 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.

[15] The real use case we had, was using <http://lucene.apache.org/solr>, where you need a separate war for each index you 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.

[16] “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.

[17] We do this here, as it makes the layout a bit easier. We usually put the project specific stuff into the build script of the respective projects.

[18] So we are well in the range of the 7 plus 2 Rule :)

Gradle Plugins

Gradle at its core intentionally provides very little for real world automation. All of the useful features, like the ability to compile Java code, are added by *plugins*. Plugins add new tasks (e.g. `JavaCompile`), domain objects (e.g. `SourceSet`), conventions (e.g. Java source is located at `src/main/java`) as well as extending core objects and objects from other plugins.

In this chapter we discuss how to use plugins and the terminology and concepts surrounding plugins.

27.1. What plugins do

Applying a plugin to a project allows the plugin to extend the project's capabilities. It can do things such as:

- Extend the Gradle model (e.g. add new DSL elements that can be configured)
- Configure the project according to conventions (e.g. add new tasks or configure sensible defaults)
- Apply specific configuration (e.g. add organizational repositories or enforce standards)

By applying plugins, rather than adding logic to the project build script, we can reap a number of benefits. Applying plugins:

- Promotes reuse and reduces the overhead of maintaining similar logic across multiple projects
- Allows a higher degree of modularization, enhancing comprehensibility and organization
- Encapsulates imperative logic and allows build scripts to be as declarative as possible

27.2. Types of plugins

There are two general types of plugins in Gradle, *script* plugins and *binary* plugins. Script plugins are additional build scripts that further configure the build and usually implement a declarative approach to manipulating the build. They are typically used within a build although they can be externalized and accessed from a remote location. Binary plugins are classes that implement the `Plugin` interface and adopt a programmatic approach to manipulating the build. Binary plugins can reside within a build script, within the project hierarchy or externally in a plugin jar.

A plugin often starts out as a script plugin (because they are easy to write) and then, as the code becomes more valuable, it's migrated to a binary plugin that can be easily tested and shared between multiple projects or organizations.

27.3. Using plugins

To use the build logic encapsulated in a plugin, Gradle needs to perform two steps. First, it needs to *resolve* the plugin, and then it needs to *apply* the plugin to the target, usually a `Project`.

Resolving a plugin means finding the correct version of the jar which contains a given plugin and adding it to the script classpath. Once a plugin is resolved, its API can be used in a build script. Script plugins are self-resolving in that they are resolved from the specific file path or URL provided when applying them. Core binary plugins provided as part of the Gradle distribution are automatically resolved.

Applying a plugin means actually executing the plugin's `Plugin.apply(T)` on the `Project` you want to enhance with the plugin. Applying plugins is *idempotent*. That is, you can safely apply any plugin multiple times without side effects.

The most common use case for using a plugin is to both resolve the plugin and apply it to the current project. Since this is such a common use case, it's recommended that build authors use the plugins DSL to both resolve and apply plugins in one step. The feature is technically still incubating, but it works well, and should be used by most users.

27.4. Script plugins

Example 27.1. Applying a script plugin

build.gradle

```
apply from: 'other.gradle'
```

Script plugins are automatically resolved and can be applied from a script on the local filesystem or at a remote location. Filesystem locations are relative to the project directory, while remote script locations are specified with an HTTP URL. Multiple script plugins (of either form) can be applied to a given target.

27.5. Binary plugins

You apply plugins by their *plugin id*, which is a globally unique identifier, or name, for plugins. Core Gradle plugins are special in that they provide short names, such as `'java'` for the core `JavaPlugin`. All other binary plugins must use the fully qualified form of the plugin id (e.g. `com.github.foo.bar`), although some legacy plugins may still utilize a short, unqualified form. Where you put the plugin id depends on whether you are using the plugins DSL or the buildscript block.

27.5.1. Locations of binary plugins

A plugin is simply any class that implements the `Plugin` interface. Gradle provides the core plugins (e.g. `JavaPlugin`) as part of its distribution which means they are automatically resolved. However, non-core binary plugins need to be resolved before they can be applied. This can be achieved in a number of ways:

- Including the plugin from the plugin portal or a custom repository using the plugins DSL (see

Section 27.5.2, “Applying plugins with the plugins DSL”).

- Including the plugin from an external jar defined as a buildscript dependency (see the section called “Applying plugins with the buildscript block”).
- Defining the plugin as a source file under the buildSrc directory in the project (see Section 43.4, “Build sources in the buildSrc project”).
- Defining the plugin as an inline class declaration inside a build script.

For more on defining your own plugins, see Chapter 41, *Writing Custom Plugins*.

27.5.2. Applying plugins with the plugins DSL

The plugins DSL is currently incubating. Please be aware that the DSL and other configuration may change in later Gradle versions.

The new plugins DSL provides a succinct and convenient way to declare plugin dependencies. It works with the Gradle plugin portal to provide easy access to both core and community plugins. The plugins DSL block configures an instance of `PluginDependenciesSpec`.

To apply a core plugin, the short name can be used:

Example 27.2. Applying a core plugin

build.gradle

```
plugins {  
    id 'java'  
}
```

To apply a community plugin from the portal, the fully qualified plugin id must be used:

Example 27.3. Applying a community plugin

build.gradle

```
plugins {  
    id "com.jfrog.bintray" version "0.4.1"  
}
```

See `PluginDependenciesSpec` for more information on using the Plugin DSL.

Limitations of the plugins DSL

This way of adding plugins to a project is much more than a more convenient syntax. The plugins DSL is processed in a way which allows Gradle to determine the plugins in use very early and very quickly. This allows Gradle to do smart things such as:

- Optimize the loading and reuse of plugin classes.
- Allow different plugins to use different versions of dependencies.
- Provide editors detailed information about the potential properties and values in the buildscript for

editing assistance.

This requires that plugins be specified in a way that Gradle can easily and quickly extract, before executing the rest of the build script. It also requires that the definition of plugins to use be somewhat static.

There are some key differences between the new plugin mechanism and the “traditional” `apply()` method mechanism. There are also some constraints, some of which are temporary limitations while the mechanism is still being developed and some are inherent to the new approach.

Constrained Syntax

The new `plugins {}` block does not support arbitrary Groovy code. It is constrained, in order to be idempotent (produce the same result every time) and side effect free (safe for Gradle to execute at any time).

The form is:

```
plugins {  
    id «plugin id» version «plugin version» [apply «false»]  
}
```

Where `«plugin version»` and `«plugin id»` must be constant, literal, strings and the `apply` statement with a boolean can be used to disable the default behavior of applying the plugin immediately (e.g. you want to apply it only in subprojects). No other statements are allowed; their presence will cause a compilation error.

The `plugins {}` block must also be a top level statement in the buildscript. It cannot be nested inside another construct (e.g. an if-statement or for-loop).

Can only be used in build scripts

The `plugins {}` block can currently only be used in a project's build script. It cannot be used in script plugins, the `settings.gradle` file or init scripts.

Future versions of Gradle will remove this restriction.

If the restrictions of the new syntax are prohibitive, the recommended approach is to apply plugins using the builds

Applying plugins to subprojects

If you have a multi-project build, you probably want to apply plugins to some or all of the subprojects in your build, but not to the root or master project. The default behavior of the `plugins {}` block is to immediately resolve and apply the plugins. But, you can use the `apply false` syntax to tell Gradle not to apply the plugin to the current project and then use `apply plugin: «plugin version»` in the `subprojects` block:

Example 27.4. Applying plugins only on certain subprojects.

settings.gradle

```
include 'helloA'
include 'helloB'
include 'goodbyeC'
```

build.gradle

```
plugins {
    id "org.gradle.sample.hello" version "1.0.0" apply false
    id "org.gradle.sample.goodbye" version "1.0.0" apply false
}

subprojects { subproject ->
    if (subproject.name.startsWith("hello")) {
        apply plugin: 'org.gradle.sample.hello'
    }
    if (subproject.name.startsWith("goodbye")) {
        apply plugin: 'org.gradle.sample.goodbye'
    }
}
```

If you then run `gradle hello` you'll see that only the `helloA` and `helloB` subprojects had the `hello` plugin applied.

```
gradle/subprojects/docs/src/samples/plugins/multiproject $> gradle hello
Parallel execution is an incubating feature.
:helloA:hello
:helloB:hello
Hello!
Hello!

BUILD SUCCESSFUL
```

Custom Plugin Repositories

The `pluginRepositories {}` DSL is currently incubating. Please be aware that the DSL and other configuration may change in later Gradle versions.

By default, the `plugins {}` DSL resolves plugins from the public Gradle Plugin Portal. Many build authors would also like to resolve plugins from private Maven or Ivy repositories because the plugins contain proprietary implementation details, or just to have more control over what plugins are available to their builds.

To specify custom plugin repositories, add a `pluginRepositories {}` block to the `settings.gradle` file:

Example 27.5. Using plugins from custom plugin repositories.

settings.gradle

```
pluginRepositories {  
    maven {  
        url 'maven-repo'  
    }  
    gradlePluginPortal()  
    ivy {  
        url 'ivy-repo'  
    }  
}
```

This tells Gradle to first look in the Maven repository at `maven-repo` when resolving plugins and then to check the Gradle Plugin Portal if the plugins are not found in the Maven repository. If you don't want the Gradle Plugin Portal to be searched, omit the `gradlePluginPortal()` line. Finally, the Ivy repository at `ivy-repo` will be checked.

The `pluginRepositories {}` block may only appear in the `settings.gradle` file, and must be the first block in the file. Custom Maven and Ivy plugin repositories must contain plugin marker artifacts in addition to the artifacts which actually implement the plugin. For more information on publishing plugins to custom repositories read Chapter 42, *The Java Gradle Plugin Development Plugin*.

See `PluginRepositoriesSpec` for complete documentation for using the `pluginRepositories {}` block.

Plugin Marker Artifacts

Since the `plugins {}` DSL block only allows for declaring plugins by their globally unique `plugin id` and `version` properties, Gradle needs a way to look up the coordinates of the plugin implementation artifact. To do so, Gradle will look for a Plugin Marker Artifact with the coordinates `plugin.id:plugin.id`. This marker needs to have a dependency on the actual plugin implementation. Publishing these markers is automated by the `java-gradle-plugin`.

For example, the following complete sample from the `sample-plugins` project shows how to publish a `org.c.plugin` and a `org.gradle.sample.goodbye` plugin to both an Ivy and Maven repository using the combination of the `java-gradle-plugin`, the `maven-publish` plugin, and the `ivy-publish` plugin.

Example 27.6. Complete Plugin Publishing Sample

build.gradle

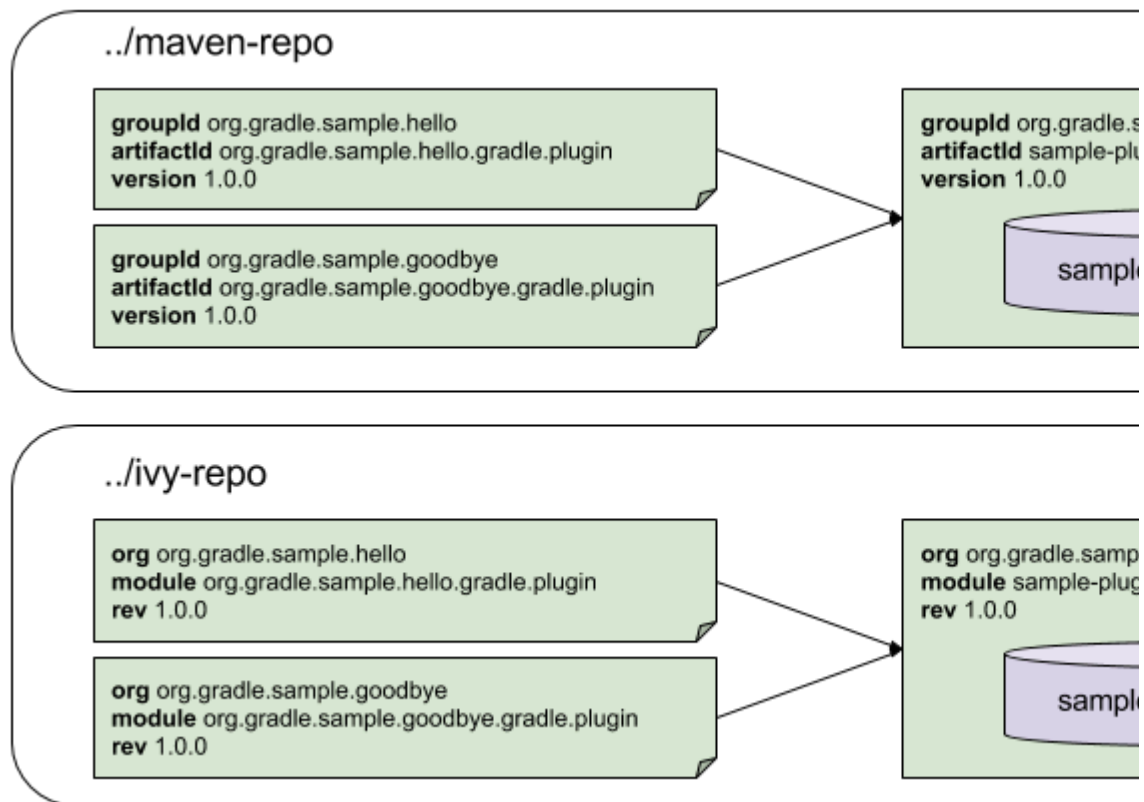
```
plugins {
    id 'java-gradle-plugin'
    id 'maven-publish'
    id 'ivy-publish'
}

group 'org.gradle.sample'
version '1.0.0'

gradlePlugin {
    plugins {
        hello {
            id = "org.gradle.sample.hello"
            implementationClass = "org.gradle.sample.hello.HelloPlugin"
        }
        goodbye {
            id = "org.gradle.sample.goodbye"
            implementationClass = "org.gradle.sample.goodbye.GoodbyePlugin"
        }
    }
}

publishing {
    repositories {
        maven {
            url "../consuming/maven-repo"
        }
        ivy {
            url "../consuming/ivy-repo"
        }
    }
}
```

Running `gradle publish` in the sample directory causes the following repo layouts to exist:



27.5.3. Legacy Plugin Application

With the introduction of the plugins DSL, users should have little reason to use the legacy method of applying plugins. It is documented here in case a build author cannot use the plugins DSL due to restrictions in how it currently works.

Applying Binary Plugins

Example 27.7. Applying a binary plugin

build.gradle

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

Plugins can be applied using a *plugin id*. In the above case, we are using the short name ‘java’ to apply the JavaPlugin.

Rather than using a plugin id, plugins can also be applied by simply specifying the class of the plugin:

Example 27.8. Applying a binary plugin by type

build.gradle

```
apply plugin: JavaPlugin
```

The JavaPlugin symbol in the above sample refers to the the JavaPlugin. This class does not strictly need to be imported as the org.gradle.api.plugins package is automatically imported in all build

scripts (see Section 18.8, “Default imports”). Furthermore, it is not necessary to append `.class` to identify a class literal in Groovy as it is in Java.

Applying plugins with the `buildscript` block

Binary plugins that have been published as external jar files can be added to a project by adding the plugin to the build script classpath and then applying the plugin. External jars can be added to the build script classpath using the `buildscript { }` block as described in Section 43.6, “External dependencies for the build script”.

Example 27.9. Applying a plugin with the `buildscript` block

build.gradle

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath "com.jfrog.bintray.gradle:gradle-bintray-plugin:0.4.1"
    }
}

apply plugin: "com.jfrog.bintray"
```

27.6. Finding community plugins

Gradle has a vibrant community of plugin developers who contribute plugins for a wide variety of capabilities. The Gradle plugin portal provides an interface for searching and exploring community plugins.

27.7. More on plugins

This chapter aims to serve as an introduction to plugins and Gradle and the role they play. For more information on the inner workings of plugins, see Chapter 41, *Writing Custom Plugins*.

Standard Gradle plugins

There are a number of plugins included in the Gradle distribution. These are listed below.

28.1. Language plugins

These plugins add support for various languages which can be compiled for and executed in the JVM.

Table 28.1. Language plugins

Plugin Id	Automatically applies	Works with	Description
java	java-base	-	Adds Java compilation, testing and bundling capabilities to a project. It serves as the basis for many of the other Gradle plugins. See also Chapter 46, <i>Java Quickstart</i> .
groovy	java, groovy-base		Adds support for building Groovy projects. See also Chapter 55, <i>Groovy Quickstart</i> .
scala	java, scala-base		Adds support for building Scala projects.
antlr	java	-	Adds support for generating parsers using Antlr.

28.2. Incubating language plugins

These plugins add support for various languages:

Table 28.2. Language plugins

Plugin Id	Automatically applies	Works with	Description
assembler	-	-	Adds native assembly language capabilities to a project.
c	-	-	Adds C source compilation capabilities to a project.
cpp	-	-	Adds C++ source compilation capabilities to a project.
objective-c	-	-	Adds Objective-C source compilation capabilities to a project.
objective-cpp	-	-	Adds Objective-C++ source compilation capabilities to a project.
windows-resources	-	-	Adds support for including Windows resources in native binaries.

28.3. Integration plugins

These plugins provide some integration with various runtime technologies.

Table 28.3. Integration plugins

Plugin Id	Automatically applies	Works with	Description
application	java, distribution		Adds tasks for running and bundling a Java project as a command-line application.
ear	-	java	Adds support for building J2EE applications.
jetty	war	-	Deploys your web application to a Jetty web container embedded in the build. See also Chapter 49, <i>Web Application Quickstart</i> . This plugin is deprecated and will be removed in Gradle 4.0. Consider using the more feature-rich Gretty plugin instead.
maven	-	java, war	Adds support for publishing artifacts to Maven repositories.
osgi	java-base	java	Adds support for building OSGi bundles.
war	java	-	Adds support for assembling web application WAR files. See also Chapter 49, <i>Web Application Quickstart</i> .

28.4. Incubating integration plugins

These plugins provide some integration with various runtime technologies.

Table 28.4. Incubating integration plugins

Plugin Id	Automatically applies	Works with	Description
distribution	-	-	Adds support for building ZIP and TAR distributions.
java-library-distribution	java,distribution		Adds support for building ZIP and TAR distributions for a Java library.
ivy-publish	-	java,war	This plugin provides a new DSL to support publishing artifacts to Ivy repositories, which improves on the existing DSL.
maven-publish	-	java,war	This plugin provides a new DSL to support publishing artifacts to Maven repositories, which improves on the existing DSL.

28.5. Software development plugins

These plugins provide help with your software development process.

Table 28.5. Software development plugins

Plugin Id	Automatically applies	Works with	Description
announce	-	-	Publish messages to your favourite platforms, such as Twitter or Growl.
build-announcements	announce	-	Sends local announcements to your desktop about interesting events in the build lifecycle.
checkstyle	java-base	-	Performs quality checks on your project's Java source files using Checkstyle and generates reports from these checks.

codenarc	groovy-base	-	Performs quality checks on your project's Groovy source files using CodeNarc and generates reports from these checks.
eclipse	-	java,groovy,scala	Generates files that are used by Eclipse IDE, thus making it possible to import the project into Eclipse. See also Chapter 46, <i>Java Quickstart</i> .
eclipse-wtp	-	ear,war	Does the same as the eclipse plugin plus generates eclipse WTP (Web Tools Platform) configuration files. After importing to eclipse your war/ear projects should be configured to work with WTP. See also Chapter 46, <i>Java Quickstart</i> .
findbugs	java-base	-	Performs quality checks on your project's Java source files using FindBugs and generates reports from these checks.
idea	-	java	Generates files that are used by IntelliJ IDEA IDE, thus making it possible to import the project into IDEA.
jdepend	java-base	-	Performs quality checks on your project's source files using JDepend and generates reports from these checks.
pmd	java-base	-	Performs quality checks on your project's Java source files using PMD and generates reports from these checks.
project-report	reporting-base	-	Generates reports containing useful information about your Gradle build.

signing	base	-	Adds the ability to digitally sign built files and artifacts.
---------	------	---	---

28.6. Incubating software development plugins

These plugins provide help with your software development process.

Table 28.6. Software development plugins

Plugin Id	Automatically applies	Works with	Description
build-dashboard	reporting-base	-	Generates build dashboard report.
build-init	wrapper	-	Adds support for initializing a new Gradle build. Handles converting a Maven build to a Gradle build.
cunit	-	-	Adds support for running CUnit tests.
jacoco	reporting-base	java	Provides integration with the JaCoCo code coverage library for Java.
visual-studio	-	native language plugins	Adds integration with Visual Studio.
wrapper	-	-	Adds a Wrapper task for generating Gradle wrapper files.
java-gradle-plugin	java		Assists with development of Gradle plugins by providing standard plugin build configuration and validation.

28.7. Base plugins

These plugins form the basic building blocks which the other plugins are assembled from. They are available for you to use in your build files, and are listed here for completeness. However, be aware that they are not yet considered part of Gradle's public API. As such, these plugins are not documented in the user guide. You might refer to their API documentation to learn more about them.

Table 28.7. Base plugins

Plugin Id	Description
base	<p>Adds the standard lifecycle tasks and configures reasonable defaults for the archive tasks:</p> <ul style="list-style-type: none"> • adds build <i>ConfigurationName</i> tasks. Those tasks assemble the artifacts belonging to the specified configuration. • adds upload <i>ConfigurationName</i> tasks. Those tasks assemble and upload the artifacts belonging to the specified configuration. • configures reasonable default values for all archive tasks (e.g. tasks that inherit from <i>AbstractArchiveTask</i>). For example, the archive tasks are tasks of types: Jar, Tar, Zip. Specifically, <i>destinationDir</i>, <i>baseName</i> and <i>version</i> properties of the archive tasks are preconfigured with defaults. This is extremely useful because it drives consistency across projects; the consistency regarding naming conventions of archives and their location after the build completed.
java-base	Adds the source sets concept to the project. Does not add any particular source sets.
groovy-base	Adds the Groovy source sets concept to the project.
scala-base	Adds the Scala source sets concept to the project.
reporting-base	Adds some shared convention properties to the project, relating to report generation.

28.8. Third party plugins

You can find a list of external plugins at the [Gradle Plugins site](#).

29

The Project Report Plugin

The Project report plugin adds some tasks to your project which generate reports containing useful information about your build. These tasks generate the same content that you get by executing the **tasks**, **dependencies**, and **properties** tasks from the command line (see Section 4.7, “Obtaining information about your build”). In contrast to the command line reports, the report plugin generates the reports into a file. There is also an aggregating task that depends on all report tasks added by the plugin.

We plan to add much more to the existing reports and create additional ones in future releases of Gradle.

29.1. Usage

To use the Project report plugin, include the following in your build script:

```
apply plugin: 'project-report'
```

29.2. Tasks

The project report plugin defines the following tasks:

Table 29.1. Project report plugin - tasks

Task name	Depends on	Type	Description
dependencyReport	-	DependencyReportTask	Generate the project dependency report.
htmlDependencyReport	-	HtmlDependencyReportTask	Generate an HTML dependency report and insight into the project or a subproject.
propertyReport	-	PropertyReportTask	Generate the project property report.
taskReport	-	TaskReportTask	Generate the project task report.
projectReport	dependencyReport, propertyReport, taskReport, htmlDependencyReport	TaskReport	Generate all project reports.

29.3. Project layout

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

29.4. Dependency management

The project report plugin does not define any dependency configurations.

29.5. Convention properties

The project report defines the following convention properties:

Table 29.2. Project report plugin - convention 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.
projects	Set<Project>	A one element set with the project the plugin was applied to.	The projects to generate the reports for.
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`.

The Build Dashboard Plugin

The build dashboard plugin is currently incubating. Please be aware that the DSL and other configuration may change in later Gradle versions.

The Build Dashboard plugin can be used to generate a single HTML dashboard that provides a single point of access to all of the reports generated by a build.

30.1. Usage

To use the Build Dashboard plugin, include the following in your build script:

Example 30.1. Using the Build Dashboard plugin

build.gradle

```
apply plugin: 'build-dashboard'
```

Applying the plugin adds the `buildDashboard` task to your project. The task aggregates the reports for all tasks that implement the `Reporting` interface from all projects in the build. It is typically only applied to the root project.

The `buildDashboard` task does not depend on any other tasks. It will only aggregate the reporting tasks that are independently being executed as part of the build run. To generate the build dashboard, simply include this task in the list of tasks to execute. For example, “**gradle buildDashboard build**” will generate a dashboard for all of the reporting tasks that are dependents of the `build` task.

30.2. Tasks

The Build Dashboard plugin adds the following task to the project:

Table 30.1. Build Dashboard plugin - tasks

Task name	Depends on	Type	Description
buildDashboard	-	GenerateBuildDashboard	Generates build dashboard report.

30.3. Project layout

The Build Dashboard plugin does not require any particular project layout.

30.4. Dependency management

The Build Dashboard plugin does not define any dependency configurations.

30.5. Configuration

You can influence the location of build dashboard plugin generation via `ReportingExtension`.

Comparing Builds

Build comparison support is an incubating feature. This means that it is incomplete and not yet at regular Gradle production quality. This also means that this Gradle User Guide chapter is a work in progress.

Gradle provides support for comparing the *outcomes* (e.g. the produced binary archives) of two builds. There are several reasons why you may want to compare the outcomes of two builds. You may want to compare:

- A build with a newer version of Gradle than it's currently using (i.e. upgrading the Gradle version).
- A Gradle build with a build executed by another tool such as Apache Ant, Apache Maven or something else (i.e. migrating to Gradle).
- The same Gradle build, with the same version, before and after a change to the build (i.e. testing build changes).

By comparing builds in these scenarios you can make an informed decision about the Gradle upgrade, migration to Gradle or build change by understanding the differences in the outcomes. The comparison process produces a HTML report outlining which outcomes were found to be identical and identifying the differences between non-identical outcomes.

31.1. Definition of terms

The following are the terms used for build comparison and their definitions.

“Build”

In the context of build comparison, a build is not necessarily a Gradle build. It can be any invocable “process” that produces observable “outcomes”. At least one of the builds in a comparison will be a Gradle build.

“Build Outcome”

Something that happens in an observable manner during a build, such as the creation of a zip file or test execution. These are the things that are compared.

“Source Build”

The build that comparisons are being made against, typically the build in its “current” state. In other words, the left hand side of the comparison.

“Target Build”

The build that is being compared to the source build, typically the “proposed” build. In other words, the right hand side of the comparison.

“Host Build”

The Gradle build that executes the comparison process. It may be the same project as either the “target” or “source” build or may be a completely separate project. It does not need to be the same Gradle version as the “source” or “target” builds. The host build must be run with Gradle 1.2 or newer.

“Compared Build Outcome”

Build outcomes that are intended to be logically equivalent in the “source” and “target” builds, and are therefore meaningfully comparable.

“Uncompared Build Outcome”

A build outcome is uncompared if a logical equivalent from the other build cannot be found (e.g. a build produces a zip file that the other build does not).

“Unknown Build Outcome”

A build outcome that cannot be understood by the host build. This can occur when the source or target build is a newer Gradle version than the host build and that Gradle version exposes new outcome types. Unknown build outcomes can be compared in so far as they can be identified to be logically equivalent to an unknown build outcome in the other build, but no meaningful comparison of what the build outcome actually is can be performed. Using the latest Gradle version for the host build will avoid encountering unknown build outcomes.

31.2. Current Capabilities

As this is an incubating feature, a limited set of the eventual functionality has been implemented at this time.

31.2.1. Supported builds

Only support for comparing Gradle builds is available at this time. Both the source and target build must execute with Gradle newer or equal to version 1 . 0. The host build must be at least version 1 . 2.

Future versions will provide support for executing builds from other build systems such as Apache Ant or Apache Maven, as well as support for executing arbitrary processes (e.g. shell script based builds)

31.2.2. Supported build outcomes

Only support for comparing build outcomes that are zip archives is supported at this time. This includes jar , war and ear archives.

Future versions will provide support for comparing outcomes such as test execution (i.e. which tests were executed, which tests failed, etc.)

31.3. Comparing Gradle Builds

The `compare-gradle-builds` plugin can be used to facilitate a comparison between two Gradle builds. The plugin adds a `CompareGradleBuilds` task named “`compareGradleBuilds`” to the project. The configuration of this task specifies what is to be compared. By default, it is configured to compare the current build with itself using the current Gradle version by executing the tasks: “`clean assemble`”.

```
apply plugin: 'compare-gradle-builds'
```

This task can be configured to change what is compared.

```
compareGradleBuilds {
    sourceBuild {
        projectDir "/projects/project-a"
        gradleVersion "1.1"
    }
    targetBuild {
        projectDir "/projects/project-b"
        gradleVersion "1.2"
    }
}
```

The example above specifies a comparison between two different projects using two different Gradle versions.

31.3.1. Trying Gradle upgrades

You can use the build comparison functionality to very quickly try a new Gradle version with your build.

To try your current build with a different Gradle version, simply add the following to the `build.gradle` of the *root project*.

```
apply plugin: 'compare-gradle-builds'

compareGradleBuilds {
    targetBuild.gradleVersion = "«gradle version»"
}
```

Then simply execute the **`compareGradleBuilds`** task. You will see the console output of the “source” and “target” builds as they are executing.

31.3.2. The comparison “result”

If there are any differences between the *compared outcomes*, the task will fail. The location of the HTML report providing insight into the comparison will be given. If all compared outcomes are found to be identical, and there are no uncompared outcomes, and there are no unknown build outcomes, the task will succeed.

You can configure the task to not fail on compared outcome differences by setting the `ignoreFailures` property to true.

```
compareGradleBuilds {  
    ignoreFailures = true  
}
```

31.3.3. Which archives are compared?

For an archive to be a candidate for comparison, it must be added as an artifact of the archives configuration. Take a look at Chapter 32, *Publishing artifacts* for more information on how to configure and add artifacts.

The archive must also have been produced by a Zip, Jar, War, Ear task. Future versions of Gradle will support increased flexibility in this area.

Publishing artifacts

This chapter describes the *original* publishing mechanism available in Gradle 1.0: in Gradle 1.3 a new mechanism for publishing was introduced. While this new mechanism is incubating and not yet complete, it introduces some new concepts and features that do (and will) make Gradle publishing even more powerful.

You can read about the new publishing plugins in Chapter 35, *Ivy Publishing (new)* and Chapter 36, *Maven Publishing (new)*. Please try them out and give us feedback.

32.1. Introduction

This chapter is about how you declare the outgoing 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 provides to the outside world. This might be a library or a ZIP distribution or any other file. A project can publish as many artifacts as it wants.

32.2. Artifacts and configurations

Like dependencies, artifacts are grouped by configurations. In fact, a configuration can contain both artifacts and dependencies at the same time.

For each configuration in your project, Gradle provides the tasks `uploadConfigurationName` and `buildConfigurationName`.^[19] Execution of these tasks will build or upload the artifacts belonging to the respective configuration.

Table 47.5, “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 `runtime` configuration in Section 32.5, “More about project libraries”. As with dependencies, you can declare as many custom configurations as you like and assign artifacts to them.

32.3. Declaring artifacts

32.3.1. Archive task artifacts

You can use an archive task to define an artifact:

Example 32.1. Defining an artifact using an archive task

build.gradle

```
task myJar(type: Jar)

artifacts {
    archives myJar
}
```

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.

32.3.2. File artifacts

You can also use a file to define an artifact:

Example 32.2. Defining an artifact using a file

build.gradle

```
def someFile = file('build/somefile.txt')

artifacts {
    archives someFile
}
```

Gradle will figure out the properties of the artifact based on the name of the file. You can customize these properties:

Example 32.3. Customizing an artifact

build.gradle

```
task myTask(type: MyTaskType) {
    destFile = file('build/somefile.txt')
}

artifacts {
    archives(myTask.destFile) {
        name 'my-artifact'
        type 'text'
        builtBy myTask
    }
}
```

There is a map-based syntax for defining an artifact using a file. The map must include a `file` entry that defines the file. The map may include other artifact properties:

Example 32.4. Map syntax for defining an artifact using a file

build.gradle

```
task generate(type: MyTaskType) {
    destFile = file('build/somefile.txt')
}

artifacts {
    archives file: generate.destFile, name: 'my-artifact', type: 'text', builtBy
}
```

32.4. Publishing artifacts

We have said that there is a specific upload task for each configuration. Before you can do an upload, you have to configure the upload task and define where to publish the artifacts to. The repositories you have defined (as described in Section 25.6, “Repositories”) are not automatically used for uploading. In fact, some of those repositories only allow downloading artifacts, not uploading. Here is an example of how you can configure the upload task of a configuration:

Example 32.5. Configuration of the upload task

build.gradle

```
repositories {
    flatDir {
        name "fileRepo"
        dirs "repo"
    }
}

uploadArchives {
    repositories {
        add project.repositories.fileRepo
        ivy {
            credentials {
                username "username"
                password "pw"
            }
            url "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 25.6.9, “More about Ivy resolvers”, you can use all the Ivy resolvers suitable for the purpose of uploading.

If an upload repository is defined with multiple patterns, Gradle must choose a pattern to use for uploading each file. By default, Gradle will upload to the pattern defined by the `url` parameter, combined with the

optional `layout` parameter. If no `url` parameter is supplied, then Gradle will use the first defined `artifactP` for uploading, or the first defined `ivyPattern` for uploading Ivy files, if this is set.

Uploading to a Maven repository is described in Section 33.6, “Interacting with Maven repositories”.

32.5. 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 `runtime` configuration for this purpose, with the implicit assumption that the `runtime` dependencies are the dependencies of the artifact you want to publish. 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 a 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 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 25.4.9, “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 33.6, “Interacting with Maven repositories”.

[19] To be exact, the Base plugin provides those tasks. This plugin is automatically applied if you use the Java plugin.

The Maven Plugin

This chapter is a work in progress

The Maven plugin adds support for deploying artifacts to Maven repositories.

33.1. Usage

To use the Maven plugin, include the following in your build script:

Example 33.1. Using the Maven plugin

build.gradle

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

33.2. Tasks

The Maven plugin defines the following tasks:

Table 33.1. Maven plugin - tasks

Task name	Depends on	Type	Description
install	All tasks that build the associated archives.	Upload	Installs the associated artifacts to the local Maven cache, including Maven metadata generation. By default the install task is associated with the <code>archives</code> configuration. This configuration has by default only the default jar as an element. To learn more about installing to the local repository, see: Section 33.6.3, “Installing to the local repository”

33.3. Dependency management

The Maven plugin does not define any dependency configurations.

33.4. Convention properties

The Maven plugin defines the following convention properties:

Table 33.2. Maven plugin - properties

Property name	Type	Default value	Description
mavenPomDir	File	<code>\${project.buildDir}/pom.xml</code>	The directory where generated POMs are written.
conf2ScopeMappings	Conf2ScopeMappingContainer	n/a	Instructions for mapping Gradle configurations to Maven scopes. The section is called “Dependency Scope Mappings”.

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

33.5. Convention methods

The maven plugin provides a factory method for creating a POM. This is useful if you need a POM without the context of uploading to a Maven repo.

Example 33.2. Creating a stand alone pom.

build.gradle

```
task writeNewPom {
    doLast {
        pom {
            project {
                inceptionYear '2008'
                licenses {
                    license {
                        name 'The Apache Software License, Version 2.0'
                        url 'http://www.apache.org/licenses/LICENSE-2.0.txt'
                        distribution 'repo'
                    }
                }
            }
        }.writeTo("$buildDir/newpom.xml")
    }
}
```

Amongst other things, Gradle supports the same builder syntax as polyglot Maven. To learn more about the Gradle Maven POM object, see [MavenPom](#). See also: [MavenPluginConvention](#)

33.6. Interacting with Maven repositories

33.6.1. 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.

33.6.2. 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 33.3. Upload of file to remote Maven repository

build.gradle

```
apply plugin: '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 more work to do if you need support for protocols other than `file`. In this case the native Maven code we delegate to needs additional libraries. Which libraries are needed depends on what protocol you plan to use. The available protocols and the corresponding libraries are listed in Table 33.3, “Protocol jars for Maven deployment” (those libraries have transitive dependencies which have transitive dependencies).[[]

^{20]} For example, to use the `ssh` protocol you can do:

Example 33.4. Upload of file via SSH

build.gradle

```
configurations {
    deployerJars
}

repositories {
    mavenCentral()
}

dependencies {
    deployerJars "org.apache.maven.wagon:wagon-ssh:2.2"
}

uploadArchives {
    repositories.mavenDeployer {
        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 bean elements to its parent, you use a closure. In the example above *repository* and *authentication* are such bean elements. Table 33.4, “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 33.3. Protocol jars for Maven deployment

Protocol	Library
http	org.apache.maven.wagon:wagon-http:2.2
ssh	org.apache.maven.wagon:wagon-ssh:2.2
ssh-external	org.apache.maven.wagon:wagon-ssh-external:2.2
ftp	org.apache.maven.wagon:wagon-ftp:2.2
webdav	org.apache.maven.wagon:wagon-webdav:1.0-beta-2
file	-

Table 33.4. 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

33.6.3. 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 `archi` 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.

33.6.4. Maven POM generation

When deploying an artifact to a Maven repository, Gradle automatically generates a POM for it. The `groupId`, `artifactId`, `version` and `packaging` elements used for the POM default to the values shown in the table below. The dependency elements are created from the project's dependency declarations.

Table 33.5. Default Values for Maven POM generation

Maven Element	Default Value
groupId	project.group
artifactId	uploadTask.repositories.mavenDeployer.pom.artifactId (if set) or archiveTask.baseName.
version	project.version
packaging	archiveTask.extension

Here, `uploadTask` and `archiveTask` refer to the tasks used for uploading and generating the archive, respectively (for example `uploadArchives` and `jar`). `archiveTask.baseName` defaults to `project.ar` which in turn defaults to `project.name`.

When you set the “`archiveTask.baseName`” property to a value other than the default, you'll also have to set `uploadTask.repositories.mavenDeployer.pom.artifactId` to the same value. Otherwise, the project at hand may be referenced with the wrong artifact ID from generated POMs for other projects in the same build.

Generated POMs can be found in `<buildDir>/poms`. They can be further customized via the `MavenPom` API. For example, 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 33.5. Customization of pom

build.gradle

```
uploadArchives {
    repositories {
        mavenDeployer {
            repository(url: "file://localhost/tmp/myRepo/")
            pom.version = '1.0Maven'
            pom.artifactId = 'myMavenName'
        }
    }
}
```

To add additional content to the POM, the `pom.project` builder can be used. With this builder, any element listed in the Maven POM reference can be added.

Example 33.6. Builder style customization of pom

build.gradle

```
uploadArchives {
    repositories {
        mavenDeployer {
            repository(url: "file://localhost/tmp/myRepo/")
            pom.project {
                licenses {
                    license {
                        name 'The Apache Software License, Version 2.0'
                        url 'http://www.apache.org/licenses/LICENSE-2.0.txt'
                        distribution 'repo'
                    }
                }
            }
        }
    }
}
```

Note: groupId, artifactId, version, and packaging should always be set directly on the pom object.

Example 33.7. Modifying auto-generated content

build.gradle

```
def installer = install.repositories.mavenInstaller
def deployer = uploadArchives.repositories.mavenDeployer

[installer, deployer]*.pom*.whenConfigured {pom ->
    pom.dependencies.find {dep -> dep.groupId == 'group3' && dep.artifactId == 'runt
}
```

If you have more than one artifact to publish, things work a little bit differently. See the section called “Multiple artifacts per project”.

To customize the settings for the Maven installer (see Section 33.6.3, “Installing to the local repository”), you can do:

Example 33.8. Customization of Maven installer

build.gradle

```
install {
    repositories.mavenInstaller {
        pom.version = '1.0Maven'
        pom.artifactId = 'myName'
    }
}
```

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 33.9. 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 `PomFilterContainer` and its associated classes.

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. You can also assign 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 33.10. Accessing a mapping configuration

build.gradle

```
task mappings {
    doLast {
        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). Per-configuration excludes are also included in the Maven POM, if they are convertible.

[20] It is planned for a future release to provide out-of-the-box support for this

34

The Signing Plugin

The signing plugin adds the ability to digitally sign built files and artifacts. These digital signatures can then be used to prove who built the artifact the signature is attached to as well as other information such as when the signature was generated.

The signing plugin currently only provides support for generating OpenPGP signatures (which is the signature format required for publication to the Maven Central Repository).

34.1. Usage

To use the Signing plugin, include the following in your build script:

Example 34.1. Using the Signing plugin

build.gradle

```
apply plugin: 'signing'
```

34.2. Signatory credentials

In order to create OpenPGP signatures, you will need a key pair (instructions on creating a key pair using the GnuPG tools can be found in the GnuPG HOWTOs). You need to provide the signing plugin with your key information, which means three things:

- The public key ID (an 8 character hexadecimal string).
- The absolute path to the secret key ring file containing your private key.
- The passphrase used to protect your private key.

These items must be supplied as the values of properties `signing.keyId`, `signing.secretKeyRingFile`, and `signing.password` respectively. Given the personal and private nature of these values, a good practice is to store them in the user `gradle.properties` file (described in Section 12.2, “Gradle properties and system properties”).

```
signing.keyId=24875D73
signing.password=secret
signing.secretKeyRingFile=/Users/me/.gnupg/secring.gpg
```

If specifying this information (especially `signing.password`) in the user `gradle.properties` file is not feasible for your environment, you can source the information however you need to and set the project properties manually.

```
import org.gradle.plugins.signing.Sign

gradle.taskGraph.whenReady { taskGraph ->
    if (taskGraph.allTasks.any { it instanceof Sign }) {
        // Use Java 6's console to read from the console (no good for
        // a CI environment)
        Console console = System.console()
        console.printf "\n\nWe have to sign some things in this build." +
            "\n\nPlease enter your signing details.\n\n"

        def id = console.readLine("PGP Key Id: ")
        def file = console.readLine("PGP Secret Key Ring File (absolute path): ")
        def password = console.readPassword("PGP Private Key Password: ")

        allprojects { ext."signing.keyId" = id }
        allprojects { ext."signing.secretKeyRingFile" = file }
        allprojects { ext."signing.password" = password }

        console.printf "\nThanks.\n\n"
    }
}
```

34.2.1. Using OpenPGP subkeys

OpenPGP supports subkeys, which are like the normal keys, except they're bound to a master key pair. One feature of OpenPGP subkeys is that they can be revoked independently of the master keys which makes key management easier. A practical case study of how subkeys can be leveraged in software development can be read on the Debian wiki.

The signing plugin supports OpenPGP subkeys out of the box. Just specify a subkey ID as the value in the `signing.keyId` property.

34.3. Specifying what to sign

As well as configuring how things are to be signed (i.e. the signatory configuration), you must also specify what is to be signed. The Signing plugin provides a DSL that allows you to specify the tasks and/or configurations that should be signed.

34.3.1. Signing Configurations

It is common to want to sign the artifacts of a configuration. For example, the Java plugin configures a jar to build and this jar artifact is added to the `archives` configuration. Using the Signing DSL, you can specify that all of the artifacts of this configuration should be signed.

Example 34.2. Signing a configuration

build.gradle

```
signing {  
    sign configurations.archives  
}
```

This will create a task (of type `Sign`) in your project named “`signArchives`”, that will build any archives artifacts (if needed) and then generate signatures for them. The signature files will be placed alongside the artifacts being signed.

Example 34.3. Signing a configuration output

Output of `gradle signArchives`

```
> gradle signArchives  
:compileJava  
:processResources  
:classes  
:jar  
:signArchives
```

BUILD SUCCESSFUL

Total time: 1 secs

34.3.2. Signing Tasks

In some cases the artifact that you need to sign may not be part of a configuration. In this case you can directly sign the task that produces the artifact to sign.

Example 34.4. Signing a task

build.gradle

```
task stuffZip (type: Zip) {  
    baseName = "stuff"  
    from "src/stuff"  
}  
  
signing {  
    sign stuffZip  
}
```

This will create a task (of type `Sign`) in your project named “`signStuffZip`”, that will build the input task's archive (if needed) and then sign it. The signature file will be placed alongside the artifact being signed.

Example 34.5. Signing a task output

Output of **gradle signStuffZip**

```
> gradle signStuffZip
:stuffZip
:signStuffZip

BUILD SUCCESSFUL

Total time: 1 secs
```

For a task to be “signable”, it must produce an archive of some type. Tasks that do this are the Tar, Zip, Jar, War and Ear tasks.

34.3.3. Conditional Signing

A common usage pattern is to only sign build artifacts under certain conditions. For example, you may not wish to sign artifacts for non release versions. To achieve this, you can specify that signing is only required under certain conditions.

Example 34.6. Conditional signing

build.gradle

```
version = '1.0-SNAPSHOT'
ext.isReleaseVersion = !version.endsWith("SNAPSHOT")

signing {
    required { isReleaseVersion && gradle.taskGraph.hasTask("uploadArchives") }
    sign configurations.archives
}
```

In this example, we only want to require signing if we are building a release version and we are going to publish it. Because we are inspecting the task graph to determine if we are going to be publishing, we must set the `signing.required` property to a closure to defer the evaluation. See `SigningExtension.setRequired(java.lang.Object)` for more information.

34.4. Publishing the signatures

When specifying what is to be signed via the Signing DSL, the resultant signature artifacts are automatically added to the `signatures` and `archives` dependency configurations. This means that if you want to upload your signatures to your distribution repository along with the artifacts you simply execute the `uploadArc` task as normal.

34.5. Signing POM files

When deploying signatures for your artifacts to a Maven repository, you will also want to sign the published POM file. The signing plugin adds a `signing.signPom()` (see: `SigningExtension.signPom(org.gradle.api.artifacts.maven.MavenDeployment, groovy.lang.Closure)`) method that can be used in the `beforeDeployment()` block in your upload task configuration.

Example 34.7. Signing a POM for deployment

build.gradle

```
uploadArchives {
    repositories {
        mavenDeployer {
            beforeDeployment { MavenDeployment deployment -> signing.signPom(deployment)
        }
    }
}
```

When signing is not required and the POM cannot be signed due to insufficient configuration (i.e. no credentials for signing) then the `signPom()` method will silently do nothing.

Ivy Publishing (new)

This chapter describes the new incubating Ivy publishing support provided by the “ivy-publish” plugin. Eventually this new publishing support will replace publishing via the Upload task.

If you are looking for documentation on the original Ivy publishing support using the Upload task please see Chapter 32, *Publishing artifacts*.

This chapter describes how to publish build artifacts in the Apache Ivy format, usually to a repository for consumption by other builds or projects. What is published is one or more artifacts created by the build, and an Ivy *module descriptor* (normally `ivy.xml`) that describes the artifacts and the dependencies of the artifacts, if any.

A published Ivy module can be consumed by Gradle (see Chapter 25, *Dependency Management*) and other tools that understand the Ivy format.

35.1. The “ivy-publish” Plugin

The ability to publish in the Ivy format is provided by the “ivy-publish” plugin.

The “publishing” plugin creates an extension on the project named “publishing” of type `PublishingExtension`. This extension provides a container of named publications and a container of named repositories. The “ivy-publish” plugin works with `IvyPublication` publications and `IvyArtifactRepository` repositories.

Example 35.1. Applying the “ivy-publish” plugin

build.gradle

```
apply plugin: 'ivy-publish'
```

Applying the “ivy-publish” plugin does the following:

- Applies the “publishing” plugin
- Establishes a rule to automatically create a `GenerateIvyDescriptor` task for each `IvyPublication` added (see Section 35.2, “Publications”).
- Establishes a rule to automatically create a `PublishToIvyRepository` task for the combination of

each `IvyPublication` added (see Section 35.2, “Publications”), with each `IvyArtifactRepository` added (see Section 35.3, “Repositories”).

35.2. Publications

If you are not familiar with project artifacts and configurations, you should read Chapter 32, *Publishing artifacts*, which introduces these concepts. This chapter also describes “publishing artifacts” using a different mechanism than what is described in this chapter. The publishing functionality described here will eventually supersede that functionality.

Publication objects describe the structure/configuration of a publication to be created. Publications are published to repositories via tasks, and the configuration of the publication object determines exactly what is published. All of the publications of a project are defined in the `PublishingExtension.getPublications()` container. Each publication has a unique name within the project.

For the “ivy-publish” plugin to have any effect, an `IvyPublication` must be added to the set of publications. This publication determines which artifacts are actually published as well as the details included in the associated Ivy module descriptor file. A publication can be configured by adding components, customizing artifacts, and by modifying the generated module descriptor file directly.

35.2.1. Publishing a Software Component

The simplest way to publish a Gradle project to an Ivy repository is to specify a `SoftwareComponent` to publish. The components presently available for publication are:

Table 35.1. Software Components

Name	Provided By	Artifacts	Dependencies
java	Java Plugin	Generated jar file	Dependencies from 'runtime' configuration
web	War Plugin	Generated war file	No dependencies

In the following example, artifacts and runtime dependencies are taken from the ``java`` component, which is added by the `Java Plugin`.

Example 35.2. Publishing a Java module to Ivy

build.gradle

```
publications {
    ivyJava(IvyPublication) {
        from components.java
    }
}
```

35.2.2. Publishing custom artifacts

It is also possible to explicitly configure artifacts to be included in the publication. Artifacts are commonly supplied as raw files, or as instances of `AbstractArchiveTask` (e.g. Jar, Zip).

For each custom artifact, it is possible to specify the name, extension, type, classifier and conf values to use for publication. Note that each artifacts must have a unique name/classifier/extension combination.

Configure custom artifacts as follows:

Example 35.3. Publishing additional artifact to Ivy

build.gradle

```
task sourceJar(type: Jar) {
    from sourceSets.main.java
    classifier "source"
}
publishing {
    publications {
        ivy(IvyPublication) {
            from components.java
            artifact(sourceJar) {
                type "source"
                conf "runtime"
            }
        }
    }
}
```

See the `IvyPublication` class in the API documentation for more detailed information on how artifacts can be customized.

35.2.3. Identity values for the published project

The generated Ivy module descriptor file contains an `<info>` element that identifies the module. The default identity values are derived from the following:

- organisation - `Project.getGroup()`
- module - `Project.getName()`
- revision - `Project.getVersion()`
- status - `Project.getStatus()`
- branch - (not set)

Overriding the default identity values is easy: simply specify the organisation, module or revision attributes when configuring the `IvyPublication`. The status and branch attributes can be set via the descriptor property (see `IvyModuleDescriptorSpec`). The descriptor property can also be used to add additional custom elements as children of the `<info>` element.

Example 35.4. customizing the publication identity

build.gradle

```
publishing {
    publications {
        ivy(IvyPublication) {
            organisation 'org.gradle.sample'
            module 'project1-sample'
            revision '1.1'
            descriptor.status = 'milestone'
            descriptor.branch = 'testing'
            descriptor.extraInfo 'http://my.namespace', 'myElement', 'Some value'

            from components.java
        }
    }
}
```

Gradle will handle any valid Unicode character for organisation, module and revision (as well as artifact name, extension and classifier). The only values that are explicitly prohibited are '\', '/' and any ISO control character. The supplied values are validated early during publication.

Certain repositories are not able to handle all supported characters. For example, the ':' character cannot be used as an identifier when publishing to a filesystem-backed repository on Windows.

35.2.4. Modifying the generated module descriptor

At times, the module descriptor file generated from the project information will need to be tweaked before publishing. The “ivy-publish” plugin provides a hook to allow such modification.

Example 35.5. Customizing the module descriptor file

build.gradle

```
publications {
    ivyCustom(IvyPublication) {
        descriptor.withXml {
            asNode().info[0].appendNode('description',
                                         'A demonstration of ivy descriptor customiza
            )
        }
    }
}
```

In this example we are simply adding a 'description' element to the generated Ivy dependency descriptor, but this hook allows you to modify any aspect of the generated descriptor. For example, you could replace the version range for a dependency with the actual version used to produce the build.

See `IvyModuleDescriptorSpec.withXml(org.gradle.api.Action)` in the API documentation for more information.

It is possible to modify virtually any aspect of the created descriptor should you need to. This means that it is also possible to modify the descriptor in such a way that it is no longer a valid Ivy module descriptor, so

care must be taken when using this feature.

The identifier (organisation, module, revision) of the published module is an exception; these values cannot be modified in the descriptor using the `withXML` hook.

35.2.5. Publishing multiple modules

Sometimes it's useful to publish multiple modules from your Gradle build, without creating a separate Gradle subproject. An example is publishing a separate API and implementation jar for your library. With Gradle this is simple:

Example 35.6. Publishing multiple modules from a single project

build.gradle

```
task apiJar(type: Jar) {
    baseName "publishing-api"
    from sourceSets.main.output
    exclude '**/impl/**'
}
publishing {
    publications {
        impl(IvyPublication) {
            organisation 'org.gradle.sample.impl'
            module 'project2-impl'
            revision '2.3'

            from components.java
        }
        api(IvyPublication) {
            organisation 'org.gradle.sample'
            module 'project2-api'
            revision '2'
        }
    }
}
```

If a project defines multiple publications then Gradle will publish each of these to the defined repositories. Each publication must be given a unique identity as described above.

35.3. Repositories

Publications are published to repositories. The repositories to publish to are defined by the `PublishingExtension.getRepositories()` container.

Example 35.7. Declaring repositories to publish to

build.gradle

```
repositories {  
    ivy {  
        // change to point to your repo, e.g. http://my.org/repo  
        url "$buildDir/repo"  
    }  
}
```

The DSL used to declare repositories for publishing is the same DSL that is used to declare repositories for dependencies (`RepositoryHandler`). However, in the context of Ivy publication only the repositories created by the `ivy()` methods can be used as publication destinations. You cannot publish an `IvyPublication` to a Maven repository for example.

35.4. Performing a publish

The “ivy-publish” plugin automatically creates a `PublishToIvyRepository` task for each `IvyPublication` and `IvyArtifactRepository` combination in the `publishing.publications` and `publishing.repositories` containers respectively.

The created task is named “`publish«PUBNAME»PublicationTo«REPONAME»Repository`”, which is “`publishIvyJavaPublicationToIvyRepository`” for this example. This task is of type `PublishToIvyRepository`.

Example 35.8. Choosing a particular publication to publish

build.gradle

```
apply plugin: 'java'
apply plugin: 'ivy-publish'

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

publishing {
    publications {
        ivyJava(IvyPublication) {
            from components.java
        }
    }
    repositories {
        ivy {
            // change to point to your repo, e.g. http://my.org/repo
            url "$buildDir/repo"
        }
    }
}
```

Output of `gradle publishIvyJavaPublicationToIvyRepository`

```
> gradle publishIvyJavaPublicationToIvyRepository
:generateDescriptorFileForIvyJavaPublication
:compileJava NO-SOURCE
:processResources NO-SOURCE
:classes UP-TO-DATE
:jar
:publishIvyJavaPublicationToIvyRepository

BUILD SUCCESSFUL

Total time: 1 secs
```

35.4.1. The “publish” lifecycle task

The “publish” plugin (that the “ivy-publish” plugin implicitly applies) adds a lifecycle task that can be used to publish all publications to all applicable repositories named “publish”.

In more concrete terms, executing this task will execute all `PublishToIvyRepository` tasks in the project. This is usually the most convenient way to perform a publish.

Example 35.9. Publishing all publications via the “publish” lifecycle task

Output of **gradle publish**

```
> gradle publish
:generateDescriptorFileForIvyJavaPublication
:compileJava NO-SOURCE
:processResources NO-SOURCE
:classes UP-TO-DATE
:jar
:publishIvyJavaPublicationToIvyRepository
:publish

BUILD SUCCESSFUL

Total time: 1 secs
```

35.5. Generating the Ivy module descriptor file without publishing

At times it is useful to generate the Ivy module descriptor file (normally `ivy.xml`) without publishing your module to an Ivy repository. Since descriptor file generation is performed by a separate task, this is very easy to do.

The “ivy-publish” plugin creates one `GenerateIvyDescriptor` task for each registered `IvyPublication`, named “`generateDescriptorFileFor«PUBNAME»Publication`”, which will be “`generateDescriptorFileForIvyJavaPublication`” for the previous example of the “ivyJava” publication.

You can specify where the generated Ivy file will be located by setting the `destination` property on the generated task. By default this file is written to “`build/publications/«PUBNAME»/ivy.xml`”.

Example 35.10. Generating the Ivy module descriptor file

build.gradle

```
model {
    tasks.generateDescriptorFileForIvyCustomPublication {
        destination = file("$buildDir/generated-ivy.xml")
    }
}
```

Output of **gradle generateDescriptorFileForIvyCustomPublication**

```
> gradle generateDescriptorFileForIvyCustomPublication
:generateDescriptorFileForIvyCustomPublication

BUILD SUCCESSFUL

Total time: 1 secs
```

The “ivy-publish” plugin leverages some experimental support for late plugin configuration, and the `GenerateIvyDescriptor` task will not be constructed until the publishing extension is configured. The simplest way to ensure that the publishing plugin is configured when you attempt to access the `GenerateIvyDescriptor` task is to place the access inside a `model` block, as the example above demonstrates.

The same applies to any attempt to access publication-specific tasks like `PublishToIvyRepository`. These tasks should be referenced from within a `model` block.

35.6. Complete example

The following example demonstrates publishing with a multi-project build. Each project publishes a Java component and a configured additional source artifact. The descriptor file is customized to include the project description for each project.

Example 35.11. Publishing a Java module

build.gradle

```
subprojects {
    apply plugin: 'java'
    apply plugin: 'ivy-publish'

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

    repositories {
        mavenCentral()
    }
    task sourceJar(type: Jar) {
        from sourceSets.main.java
        classifier "source"
    }
}

project(":project1") {
    description = "The first project"

    dependencies {
        compile 'junit:junit:4.12', project(':project2')
    }
}

project(":project2") {
    description = "The second project"

    dependencies {
        compile 'commons-collections:commons-collections:3.2.2'
    }
}

subprojects {
    publishing {
        repositories {
            ivy {
                // change to point to your repo, e.g. http://my.org/repo
                url "${rootProject.buildDir}/repo"
            }
        }
        publications {
            ivy(IvyPublication) {
                from components.java
                artifact(sourceJar) {
                    type "source"
                    conf "runtime"
                }
                descriptor.withXml {
                    asNode().info[0].appendNode('description', description)
                }
            }
        }
    }
}
```

The result is that the following artifacts will be published for each project:

- The Ivy module descriptor file: “ivy-1.0.xml”.
- The primary “jar” artifact for the Java component: “project1-1.0.jar”.
- The source “jar” artifact that has been explicitly configured: “project1-1.0-source.jar”.

When project1 is published, the module descriptor (i.e. the ivy.xml file) that is produced will look like:

Example 35.12. Example generated ivy.xml

output-ivy.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<ivy-module version="2.0">
  <info organisation="org.gradle.sample" module="project1">
    <description>The first project</description>
  </info>
  <configurations>
    <conf name="compile" visibility="public"/>
    <conf name="default" visibility="public" extends="runtime,compile"/>
    <conf name="runtime" visibility="public"/>
  </configurations>
  <publications>
    <artifact name="project1" type="jar" ext="jar" conf="runtime"/>
    <artifact name="project1" type="source" ext="jar" conf="runtime" m:classifier="source"/>
  </publications>
  <dependencies>
    <dependency org="junit" name="junit" rev="4.12" conf="runtime->default"/>
    <dependency org="org.gradle.sample" name="project2" rev="1.0" conf="runtime->default"/>
  </dependencies>
</ivy-module>
```

Note that «PUBLICATION-TIME-STAMP» in this example Ivy module descriptor will be the timestamp of when the descriptor was generated.

35.7. Future features

The “ivy-publish” plugin functionality as described above is incomplete, as the feature is still incubating. In upcoming Gradle releases, the functionality will be expanded to include (but not limited to):

- Convenient customization of module attributes (module, organisation etc.)
- Convenient customization of dependencies reported in module descriptor.
- Multiple discrete publications per project

Maven Publishing (new)

This chapter describes the new incubating Maven publishing support provided by the “maven-publish” plugin. Eventually this new publishing support will replace publishing via the Upload task.

If you are looking for documentation on the original Maven publishing support using the Upload task please see Chapter 32, *Publishing artifacts*.

This chapter describes how to publish build artifacts to an Apache Maven Repository. A module published to a Maven repository can be consumed by Maven, Gradle (see Chapter 25, *Dependency Management*) and other tools that understand the Maven repository format.

36.1. The “maven-publish” Plugin

The ability to publish in the Maven format is provided by the “maven-publish” plugin.

The “publishing” plugin creates an extension on the project named “publishing” of type `PublishingExtension`. This extension provides a container of named publications and a container of named repositories. The “maven-publish” plugin works with `MavenPublication` publications and `MavenArtifactRepository` repositories.

Example 36.1. Applying the 'maven-publish' plugin

build.gradle

```
apply plugin: 'maven-publish'
```

Applying the “maven-publish” plugin does the following:

- Applies the “publishing” plugin
- Establishes a rule to automatically create a `GenerateMavenPom` task for each `MavenPublication` added (see Section 36.2, “Publications”).
- Establishes a rule to automatically create a `PublishToMavenRepository` task for the combination of each `MavenPublication` added (see Section 36.2, “Publications”), with each `MavenArtifactRepository` added (see Section 36.3, “Repositories”).
- Establishes a rule to automatically create a `PublishToMavenLocal` task for each `MavenPublication` added (see Section 36.2, “Publications”).

36.2. Publications

If you are not familiar with project artifacts and configurations, you should read the Chapter 32, *Publishing artifacts* that introduces these concepts. This chapter also describes “publishing artifacts” using a different mechanism than what is described in this chapter. The publishing functionality described here will eventually supersede that functionality.

Publication objects describe the structure/configuration of a publication to be created. Publications are published to repositories via tasks, and the configuration of the publication object determines exactly what is published. All of the publications of a project are defined in the `PublishingExtension.getPublications()` container. Each publication has a unique name within the project.

For the “maven-publish” plugin to have any effect, a `MavenPublication` must be added to the set of publications. This publication determines which artifacts are actually published as well as the details included in the associated POM file. A publication can be configured by adding components, customizing artifacts, and by modifying the generated POM file directly.

36.2.1. Publishing a Software Component

The simplest way to publish a Gradle project to a Maven repository is to specify a `SoftwareComponent` to publish. The components presently available for publication are:

Table 36.1. Software Components

Name	Provided By	Artifacts	Dependencies
java	Chapter 47, <i>The Java Plugin</i>	Generated jar file	Dependencies from 'runtime' configuration
web	Chapter 50, <i>The War Plugin</i>	Generated war file	No dependencies

In the following example, artifacts and runtime dependencies are taken from the ``java`` component, which is added by the `Java Plugin`.

Example 36.2. Adding a `MavenPublication` for a Java component

build.gradle

```
publishing {
    publications {
        mavenJava(MavenPublication) {
            from components.java
        }
    }
}
```


36.2.2. Publishing custom artifacts

It is also possible to explicitly configure artifacts to be included in the publication. Artifacts are commonly supplied as raw files, or as instances of `AbstractArchiveTask` (e.g. Jar, Zip).

For each custom artifact, it is possible to specify the `extension` and `classifier` values to use for publication. Note that only one of the published artifacts can have an empty classifier, and all other artifacts must have a unique classifier/extension combination.

Configure custom artifacts as follows:

Example 36.3. Adding additional artifact to a `MavenPublication`

build.gradle

```
task sourceJar(type: Jar) {
    from sourceSets.main.allJava
}

publishing {
    publications {
        mavenJava(MavenPublication) {
            from components.java

            artifact sourceJar {
                classifier "sources"
            }
        }
    }
}
```

See the `MavenPublication` class in the API documentation for more information about how artifacts can be customized.

36.2.3. Identity values in the generated POM

The attributes of the generated POM file will contain identity values derived from the following project properties:

- `groupId` - `Project.getGroup()`
- `artifactId` - `Project.getName()`
- `version` - `Project.getVersion()`

Overriding the default identity values is easy: simply specify the `groupId`, `artifactId` or `version` attributes when configuring the `MavenPublication`.

Example 36.4. customizing the publication identity

build.gradle

```
publishing {
    publications {
        maven(MavenPublication) {
            groupId 'org.gradle.sample'
            artifactId 'project1-sample'
            version '1.1'

            from components.java
        }
    }
}
```

Maven restricts 'groupId' and 'artifactId' to a limited character set ([A-Za-z0-9_\\-\\.]+) and Gradle enforces this restriction. For 'version' (as well as artifact 'extension' and 'classifier'), Gradle will handle any valid Unicode character.

The only Unicode values that are explicitly prohibited are '\\', '/' and any ISO control character. Supplied values are validated early in publication.

Certain repositories will not be able to handle all supported characters. For example, the ':' character cannot be used as an identifier when publishing to a filesystem-backed repository on Windows.

36.2.4. Modifying the generated POM

The generated POM file may need to be tweaked before publishing. The “maven-publish” plugin provides a hook to allow such modification.

Example 36.5. Modifying the POM file

build.gradle

```
publications {
    mavenCustom(MavenPublication) {
        pom.withXml {
            asNode().appendNode('description',
                                'A demonstration of maven POM customization')
        }
    }
}
```

In this example we are adding a 'description' element for the generated POM. With this hook, you can modify any aspect of the POM. For example, you could replace the version range for a dependency with the actual version used to produce the build.

See `MavenPom.withXml(org.gradle.api.Action)` in the API documentation for more information.

It is possible to modify virtually any aspect of the created POM. This means that it is also possible to modify the POM in such a way that it is no longer a valid Maven POM, so care must be taken when using this feature.

The identifier (groupId, artifactId, version) of the published module is an exception; these values cannot be modified in the POM using the `withXML` hook.

36.2.5. Publishing multiple modules

Sometimes it's useful to publish multiple modules from your Gradle build, without creating a separate Gradle subproject. An example is publishing a separate API and implementation jar for your library. With Gradle this is simple:

Example 36.6. Publishing multiple modules from a single project

build.gradle

```
task apiJar(type: Jar) {
    baseName "publishing-api"
    from sourceSets.main.output
    exclude '**/impl/**'
}

publishing {
    publications {
        impl(MavenPublication) {
            groupId 'org.gradle.sample.impl'
            artifactId 'project2-impl'
            version '2.3'

            from components.java
        }
        api(MavenPublication) {
            groupId 'org.gradle.sample'
            artifactId 'project2-api'
            version '2'

            artifact apiJar
        }
    }
}
```

If a project defines multiple publications then Gradle will publish each of these to the defined repositories. Each publication must be given a unique identity as described above.

36.3. Repositories

Publications are published to repositories. The repositories to publish to are defined by the `PublishingExtension.getRepositories()` container.

Example 36.7. Declaring repositories to publish to

build.gradle

```
publishing {
    repositories {
        maven {
            // change to point to your repo, e.g. http://my.org/repo
            url "$buildDir/repo"
        }
    }
}
```

The DSL used to declare repositories for publication is the same DSL that is used to declare repositories to consume dependencies from, `RepositoryHandler`. However, in the context of Maven publication only `MavenArtifactRepository` repositories can be used for publication.

36.4. Performing a publish

The “maven-publish” plugin automatically creates a `PublishToMavenRepository` task for each `MavenPublication` and `MavenArtifactRepository` combination in the `publishing.publicatio` and `publishing.repositories` containers respectively.

The created task is named “publish«*PUBNAME*»PublicationTo«*REPONAME*»Repository”.

Example 36.8. Publishing a project to a Maven repository

build.gradle

```
apply plugin: 'java'
apply plugin: 'maven-publish'

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

publishing {
    publications {
        mavenJava(MavenPublication) {
            from components.java
        }
    }
}

publishing {
    repositories {
        maven {
            // change to point to your repo, e.g. http://my.org/repo
            url "$buildDir/repo"
        }
    }
}
```

Output of **gradle publish**

```
> gradle publish
:generatePomFileForMavenJavaPublication
:compileJava
:processResources NO-SOURCE
:classes
:jar
:publishMavenJavaPublicationToMavenRepository
:publish
```

BUILD SUCCESSFUL

Total time: 1 secs

In this example, a task named “publishMavenJavaPublicationToMavenRepository” is created, which is of type `PublishToMavenRepository`. This task is wired into the `publish` lifecycle task. Executing “gradle publish” builds the POM file and all of the artifacts to be published, and transfers them to the repository.

36.5. Publishing to Maven Local

For integration with a local Maven installation, it is sometimes useful to publish the module into the local `.m2` repository. In Maven parlance, this is referred to as ‘installing’ the module. The “maven-publish” plugin makes this easy to do by automatically creating a `PublishToMavenLocal` task for each `MavenPublication` in the `publishing.publications` container. Each of these tasks is wired into the `publishToMavenLocal` lifecycle task. You do not need to have `‘mavenLocal’` in your `‘publishing.repositories’` section.

The created task is named “publish«*PUBNAME*»PublicationToMavenLocal”.

Example 36.9. Publish a project to the Maven local repository

Output of **gradle publishToMavenLocal**

```
> gradle publishToMavenLocal
:generatePomFileForMavenJavaPublication
:compileJava
:processResources NO-SOURCE
:classes
:jar
:publishMavenJavaPublicationToMavenLocal
:publishToMavenLocal
```

BUILD SUCCESSFUL

Total time: 1 secs

The resulting task in this example is named “publishMavenJavaPublicationToMavenLocal”. This task is wired into the `publishToMavenLocal` lifecycle task. Executing “`gradle publishToMavenLocal`” builds the POM file and all of the artifacts to be published, and “installs” them into the local Maven repository.

36.6. Generating the POM file without publishing

At times it is useful to generate a Maven POM file for a module without actually publishing. Since POM generation is performed by a separate task, it is very easy to do so.

The task for generating the POM file is of type `GenerateMavenPom`, and it is given a name based on the name of the publication: “generatePomFileFor«*PUBNAME*»Publication”. So in the example below, where the publication is named “mavenCustom”, the task will be named “generatePomFileForMavenCustomPublication”.

Example 36.10. Generate a POM file without publishing

build.gradle

```
model {
    tasks.generatePomFileForMavenCustomPublication {
        destination = file("$buildDir/generated-pom.xml")
    }
}
```

Output of **gradle generatePomFileForMavenCustomPublication**

```
> gradle generatePomFileForMavenCustomPublication
:generatePomFileForMavenCustomPublication
```

BUILD SUCCESSFUL

Total time: 1 secs

All details of the publishing model are still considered in POM generation, including components, custom artifacts, and any modifications made via `pom.withXml`.

The “maven-publish” plugin leverages some experimental support for late plugin configuration, and any `GenerateMavenPom` tasks will not be constructed until the publishing extension is configured. The simplest way to ensure that the publishing plugin is configured when you attempt to access the `GenerateMavenPom` task is to place the access inside a `model` block, as the example above demonstrates.

The same applies to any attempt to access publication-specific tasks like `PublishToMavenRepository`. These tasks should be referenced from within a `model` block.

The Distribution Plugin

The distribution plugin is currently incubating. Please be aware that the DSL and other configuration may change in later Gradle versions.

The distribution plugin facilitates building archives that serve as distributions of the project. Distribution archives typically contain the executable application and other supporting files, such as documentation.

37.1. Usage

To use the distribution plugin, include the following in your build script:

Example 37.1. Using the distribution plugin

build.gradle

```
apply plugin: 'distribution'
```

The plugin adds an extension named “distributions” of type `DistributionContainer` to the project. It also creates a single distribution in the distributions container extension named “main”. If your build only produces one distribution you only need to configure this distribution (or use the defaults).

You can run “**gradle distZip**” to package the main distribution as a ZIP, or “**gradle distTar**” to create a TAR file. To build both types of archives just run **gradle assembleDist**. The files will be created at “`$buildDir/distributions/$project.name-$project.version.«ext»`”.

You can run “**gradle installDist**” to assemble the uncompressed distribution into “`$buildDir/install`”.

37.2. Tasks

The Distribution plugin adds the following tasks to the project:

Table 37.1. Distribution plugin - tasks

Task name	Depends on	Type	Description
distZip	-	Zip	Creates a ZIP archive of the distribution contents
distTar	-	Tar	Creates a TAR archive of the distribution contents
assembleDist	distTar, distZip	Task	Creates ZIP and TAR archives with the distribution contents
installDist	-	Sync	Assembles the distribution content and installs it on the current machine

For each extra distribution set you add to the project, the distribution plugin adds the following tasks:

Table 37.2. Multiple distributions - tasks

Task name	Depends on	Type	Description
<code>\${distribution.name}DistZip</code>	-	Zip	Creates a ZIP archive of the distribution contents
<code>\${distribution.name}DistTar</code>	-	Tar	Creates a TAR archive of the distribution contents
<code>assemble\${distribution.name.capitalize()}Dist</code>	<code>\${distribution.name}DistTar, \${distribution.name}DistZip</code>	Task	Assembles all distribution archives
<code>install\${distribution.name.capitalize()}Dist</code>		Sync	Assembles the distribution content and installs it on the current machine

Example 37.2. Adding extra distributions

build.gradle

```
apply plugin: 'distribution'

version = '1.2'
distributions {
    custom {}
}
```

This will add following tasks to the project:

- customDistZip
- customDistTar
- assembleCustomDist
- installCustomDist

Given that the project name is “myproject” and version “1.2”, running “**gradle customDistZip**” will produce a ZIP file named “myproject-custom-1.2.zip”.

Running “**gradle installCustomDist**” will install the distribution contents into “*\$buildDir/install*”.

37.3. Distribution contents

All of the files in the “src/*\$distribution.name*/dist” directory will automatically be included in the distribution. You can add additional files by configuring the `Distribution` object that is part of the container.

Example 37.3. Configuring the main distribution

build.gradle

```
apply plugin: 'distribution'

distributions {
    main {
        baseName = 'someName'
        contents {
            from { 'src/readme' }
        }
    }
}

apply plugin: 'maven'

uploadArchives {
    repositories {
        mavenDeployer {
            repository(url: "file://some/repo")
        }
    }
}
```

In the example above, the content of the “src/readme” directory will be included in the distribution (along with the files in the “src/main/dist” directory which are added by default).

The “baseName” property has also been changed. This will cause the distribution archives to be created with a different name.

37.4. Publishing distributions

The distribution plugin adds the distribution archives as candidate for default publishing artifacts. With the maven plugin applied the distribution zip file will be published when running uploadArchives if no other default artifact is configured

Example 37.4. publish main distribution

build.gradle

```
apply plugin: 'maven'

uploadArchives {
    repositories {
        mavenDeployer {
            repository(url: "file://some/repo")
        }
    }
}
```

38

The Announce Plugin

The Gradle announce plugin allows you to send custom announcements during a build. The following notification systems are supported:

- Twitter
- notify-send (Ubuntu)
- Snarl (Windows)
- Growl (Mac OS X)

38.1. Usage

To use the announce plugin, apply it to your build script:

Example 38.1. Using the announce plugin

build.gradle

```
apply plugin: 'announce'
```

Next, configure your notification service(s) of choice (see table below for which configuration properties are available):

Example 38.2. Configure the announce plugin

build.gradle

```
announce {  
    username = 'myId'  
    password = 'myPassword'  
}
```

Finally, send announcements with the `announce` method:

Example 38.3. Using the announce plugin

build.gradle

```
task helloWorld {
    doLast {
        println "Hello, world!"
    }
}

helloWorld.doLast {
    announce.announce("helloWorld completed!", "twitter")
    announce.announce("helloWorld completed!", "local")
}
```

The announce method takes two String arguments: The message to be sent, and the notification service to be used. The following table lists supported notification services and their configuration properties.

Table 38.1. Announce Plugin Notification Services

Notification Service	Operating System	Configuration Properties	Further Information
twitter	Any	username, password	
snarl	Windows		
growl	Mac OS X		
notify-send	Ubuntu		Requires the notify-send package to be installed. Use <code>sudo apt-get install libnotify-bin</code> to install it.
local	Windows, Mac OS X, Ubuntu		Automatically chooses between snarl, growl, and notify-send depending on the current operating system.

38.2. Configuration

See the `AnnouncePluginExtension` class in the API documentation.

The Build Announcements Plugin

The build announcements plugin is currently incubating. Please be aware that the DSL and other configuration may change in later Gradle versions.

The build announcements plugin uses the announce plugin to send local announcements on important events in the build.

39.1. Usage

To use the build announcements plugin, include the following in your build script:

Example 39.1. Using the build announcements plugin

build.gradle

```
apply plugin: 'build-announcements'
```

That's it. If you want to tweak where the announcements go, you can configure the announce plugin to change the local announcer.

You can also apply the plugin from an init script:

Example 39.2. Using the build announcements plugin from an init script

init.gradle

```
rootProject {  
    apply plugin: 'build-announcements'  
}
```

Part IV. Extending the build

Writing Custom Task Classes

Gradle supports two 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 16, *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 19, *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 the task and configure the task using its properties. In this way, enhanced tasks let you reuse a piece of behaviour in many different places, possibly across different builds.

The behaviour and properties of an enhanced task is defined by the task's class. When you declare an enhanced task, you specify the type, or class of the task.

Implementing your own custom task class in Gradle is easy. You can implement a custom task class in pretty much any language you like, provided it ends up compiled to bytecode. In our examples, we are going to use Groovy as the implementation language, but you could use, for example, Java or Scala. In general, using Groovy is the easiest option, because the Gradle API is designed to work well with Groovy.

40.1. Packaging a task class

There are several places where you can put the source for the task class.

Build script

You can include the task class directly in the build script. This has the benefit that the task class is automatically compiled and included in the classpath of the build script without you having to do anything. However, the task class is not visible outside the build script, and so you cannot reuse the task class outside the build script it is defined in.

buildSrc project

You can put the source for the task class in the `rootProjectDir/buildSrc/src/main/groovy` directory. Gradle will take care of compiling and testing the task class and making it available on the classpath of the build script. The task class is visible to every build script used by the build. However, it is not visible outside the build, and so you cannot reuse the task class outside the build it is defined in. Using the `buildSrc` project approach separates the task declaration - that is, what the task should do - from the task implementation - that is, how the task does it.

See Chapter 43, *Organizing Build Logic* for more details about the `buildSrc` project.

Standalone project

You can create a separate project for your task class. This project produces and publishes a JAR which you can then use in multiple builds and share with others. Generally, this JAR might include some custom plugins, or bundle several related task classes into a single library. Or some combination of the two.

In our examples, we will start with the task class in the build script, to keep things simple. Then we will look at creating a standalone project.

40.2. Writing a simple task class

To implement a custom task class, you extend `DefaultTask`.

Example 40.1. Defining a custom task

build.gradle

```
class GreetingTask 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 40.2. A hello world task

build.gradle

```
task hello(type: GreetingTask)  
  
class GreetingTask extends DefaultTask {  
    @TaskAction  
    def greet() {  
        println 'hello from GreetingTask'  
    }  
}
```

Output of **gradle -q hello**

```
> gradle -q hello  
hello from GreetingTask
```

Let's add a property to the task, so we can customize it. Tasks are simply POGOs, and when you declare a task, you can set the properties or call methods on the task object. Here we add a `greeting` property, and set the value when we declare the `greeting` task.

Example 40.3. A customizable hello world task

build.gradle

```
// Use the default greeting
task hello(type: GreetingTask)

// Customize the greeting
task greeting(type: GreetingTask) {
    greeting = 'greetings from GreetingTask'
}

class GreetingTask extends DefaultTask {
    String greeting = 'hello from GreetingTask'

    @TaskAction
    def greet() {
        println greeting
    }
}
```

Output of **gradle -q hello greeting**

```
> gradle -q hello greeting
hello from GreetingTask
greetings from GreetingTask
```

40.3. A standalone project

Now we will move our task to a standalone project, so we can publish it and share it with others. This project is simply a Groovy project that produces a JAR containing the task class. Here is a simple build script for the project. It applies the Groovy plugin, and adds the Gradle API as a compile-time dependency.

Example 40.4. A build for a custom task

build.gradle

```
apply plugin: 'groovy'

dependencies {
    compile gradleApi()
    compile localGroovy()
}
```

Note: The code for this example can be found at **samples/customPlugin/plugin** in the ‘-all’ distribution of Gradle.

We just follow the convention for where the source for the task class should go.

Example 40.5. A custom task

src/main/groovy/org/gradle/GreetingTask.groovy

```
package org.gradle

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

class GreetingTask extends DefaultTask {
    String greeting = 'hello from GreetingTask'

    @TaskAction
    def greet() {
        println greeting
    }
}
```

40.3.1. Using your task class in another project

To use a task class in a build script, you need to add the class to the build script's classpath. To do this, you use a `buildscript { }` block, as described in Section 43.6, “External dependencies for the build script”. The following example shows how you might do this when the JAR containing the task class has been published to a local repository:

Example 40.6. Using a custom task in another project

build.gradle

```
buildscript {
    repositories {
        maven {
            url uri('../repo')
        }
    }
    dependencies {
        classpath group: 'org.gradle', name: 'customPlugin',
            version: '1.0-SNAPSHOT'
    }
}

task greeting(type: org.gradle.GreetingTask) {
    greeting = 'howdy!'
}
```

40.3.2. Writing tests for your task class

You can use the `ProjectBuilder` class to create `Project` instances to use when you test your task class.

Example 40.7. Testing a custom task

`src/test/groovy/org/gradle/GreetingTaskTest.groovy`

```
class GreetingTaskTest {
    @Test
    public void canAddTaskToProject() {
        Project project = ProjectBuilder.builder().build()
        def task = project.task('greeting', type: GreetingTask)
        assertTrue(task instanceof GreetingTask)
    }
}
```

40.4. Incremental tasks

Incremental tasks are an incubating feature.

Since the introduction of the implementation described above (early in the Gradle 1.6 release cycle), discussions within the Gradle community have produced superior ideas for exposing the information about changes to task implementors to what is described below. As such, the API for this feature will almost certainly change in upcoming releases. However, please do experiment with the current implementation and share your experiences with the Gradle community.

The feature incubation process, which is part of the Gradle feature lifecycle (see Appendix C, *The Feature Lifecycle*), exists for this purpose of ensuring high quality final implementations through incorporation of early user feedback.

With Gradle, it's very simple to implement a task that gets skipped when all of its inputs and outputs are up to date (see Section 19.9, “Up-to-date checks (AKA Incremental Build)”). However, there are times when only a few input files have changed since the last execution, and you'd like to avoid reprocessing all of the unchanged inputs. This can be particularly useful for a transformer task, that converts input files to output files on a 1:1 basis.

If you'd like to optimise your build so that only out-of-date inputs are processed, you can do so with an [incremental task](#).

40.4.1. Implementing an incremental task

For a task to process inputs incrementally, that task must contain an [incremental task action](#). This is a task action method that contains a single `IncrementalTaskInputs` parameter, which indicates to Gradle that the action will process the changed inputs only.

The incremental task action may supply an `IncrementalTaskInputs.outOfDate(org.gradle.api.Action)` action for processing any input file that is out-of-date, and a `IncrementalTaskInputs.removed(org.gradle.api.Action)` action that executes for any input file that has been removed since the previous execution.

Example 40.8. Defining an incremental task action

build.gradle

```
class IncrementalReverseTask extends DefaultTask {
    @InputDirectory
    def File inputDir

    @OutputDirectory
    def File outputDir

    @Input
    def inputProperty

    @TaskAction
    void execute(IncrementalTaskInputs inputs) {
        println inputs.incremental ? "CHANGED inputs considered out of date"
                                   : "ALL inputs considered out of date"

        if (!inputs.incremental)
            project.delete(outputDir.listFiles())

        inputs.outOfDate { change ->
            println "out of date: ${change.file.name}"
            def targetFile = new File(outputDir, change.file.name)
            targetFile.text = change.file.text.reverse()
        }

        inputs.removed { change ->
            println "removed: ${change.file.name}"
            def targetFile = new File(outputDir, change.file.name)
            targetFile.delete()
        }
    }
}
```

Note: The code for this example can be found at `samples/userguide/tasks/incrementalTask` in the ‘-all’ distribution of Gradle.

If for some reason the task is not run incremental, e.g. by running with `--rerun-tasks`, only the `outOfDate` action is executed, even if there were deleted input files. You should consider handling this case at the beginning, as is done in the example above.

For a simple transformer task like this, the task action simply needs to generate output files for any out-of-date inputs, and delete output files for any removed inputs.

A task may only contain a single incremental task action.

40.4.2. Which inputs are considered out of date?

When Gradle has history of a previous task execution, and the only changes to the task execution context since that execution are to input files, then Gradle is able to determine which input files need to be reprocessed by the task. In this case, the `IncrementalTaskInputs.outOfDate(org.gradle.api.Action)` action will be executed for any input file that was added or modified, and the `IncrementalTaskInputs.removed(org.gradle.api.Action)` action will be executed for any removed input file.

However, there are many cases where Gradle is unable to determine which input files need to be reprocessed. Examples include:

- There is no history available from a previous execution.
- You are building with a different version of Gradle. Currently, Gradle does not use task history from a different version.
- An `upToDateWhen` criteria added to the task returns `false`.
- An input property has changed since the previous execution.
- One or more output files have changed since the previous execution.

In any of these cases, Gradle will consider all of the input files to be `outOfDate`. The `IncrementalTaskInputs.outOfDate(org.gradle.api.Action)` action will be executed for every input file, and the `IncrementalTaskInputs.removed(org.gradle.api.Action)` action will not be executed at all.

You can check if Gradle was able to determine the incremental changes to input files with `IncrementalTaskInputs.isIncremental()`.

40.4.3. An incremental task in action

Given the incremental task implementation above, we can explore the various change scenarios by example. Note that the various mutation tasks ('updateInputs', 'removeInput', etc) are only present for demonstration purposes: these would not normally be part of your build script.

First, consider the `IncrementalReverseTask` executed against a set of inputs for the first time. In this case, all inputs will be considered “out of date”:

Example 40.9. Running the incremental task for the first time

build.gradle

```
task incrementalReverse(type: IncrementalReverseTask) {  
    inputDir = file('inputs')  
    outputDir = file("$buildDir/outputs")  
    inputProperty = project.properties['taskInputProperty'] ?: "original"  
}
```

Build layout

```
incrementalTask/  
  build.gradle  
  inputs/  
    1.txt  
    2.txt  
    3.txt
```

Output of **gradle -q incrementalReverse**

```
> gradle -q incrementalReverse  
ALL inputs considered out of date  
out of date: 1.txt  
out of date: 2.txt  
out of date: 3.txt
```

Naturally when the task is executed again with no changes, then the entire task is up to date and no files are reported to the task action:

Example 40.10. Running the incremental task with unchanged inputs

Output of **gradle -q incrementalReverse**

```
> gradle -q incrementalReverse
```

When an input file is modified in some way or a new input file is added, then re-executing the task results in those files being reported to `IncrementalTaskInputs.outOfDate(org.gradle.api.Action):`

Example 40.11. Running the incremental task with updated input files

build.gradle

```
task updateInputs() {  
    doLast {  
        file('inputs/1.txt').text = "Changed content for existing file 1."  
        file('inputs/4.txt').text = "Content for new file 4."  
    }  
}
```

Output of **gradle -q updateInputs incrementalReverse**

```
> gradle -q updateInputs incrementalReverse  
CHANGED inputs considered out of date  
out of date: 1.txt  
out of date: 4.txt
```

When an existing input file is removed, then re-executing the task results in that file being reported to `IncrementalTaskInputs.removed(org.gradle.api.Action)`:

Example 40.12. Running the incremental task with an input file removed

build.gradle

```
task removeInput() {
    doLast {
        file('inputs/3.txt').delete()
    }
}
```

Output of **gradle -q removeInput incrementalReverse**

```
> gradle -q removeInput incrementalReverse
CHANGED inputs considered out of date
removed: 3.txt
```

When an output file is deleted (or modified), then Gradle is unable to determine which input files are out of date. In this case, all input files are reported to the `IncrementalTaskInputs.outOfDate(org.gradle.api.Action)` action, and no input files are reported to the `IncrementalTaskInputs.removed(org.gradle.api.Action)` action:

Example 40.13. Running the incremental task with an output file removed

build.gradle

```
task removeOutput() {
    doLast {
        file("$buildDir/outputs/1.txt").delete()
    }
}
```

Output of **gradle -q removeOutput incrementalReverse**

```
> gradle -q removeOutput incrementalReverse
ALL inputs considered out of date
out of date: 1.txt
out of date: 2.txt
out of date: 3.txt
```

When a task input property is modified, Gradle is unable to determine how this property impacted the task outputs, so all input files are assumed to be out of date. So similar to the changed output file example, all input files are reported to the `IncrementalTaskInputs.outOfDate(org.gradle.api.Action)` action, and no input files are reported to the `IncrementalTaskInputs.removed(org.gradle.api.Action)` action:

Example 40.14. Running the incremental task with an input property changed

Output of **gradle -q -PtaskInputProperty=changed incrementalReverse**

```
> gradle -q -PtaskInputProperty=changed incrementalReverse
ALL inputs considered out of date
out of date: 1.txt
out of date: 2.txt
out of date: 3.txt
```

Writing Custom Plugins

A Gradle plugin packages up reusable pieces 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 share it with others.

You can implement a custom plugin in any language you like, provided the implementation ends up compiled as bytecode. For the examples here, we are going to use Groovy as the implementation language. You could use Java or Scala instead, if you want.

41.1. Packaging a plugin

There are several places where you can put the source for the plugin.

Build script

You can include the source for the plugin directly in the build script. This has the benefit that the plugin is automatically compiled and included in the classpath of the build script without you having to do anything. However, the plugin is not visible outside the build script, and so you cannot reuse the plugin outside the build script it is defined in.

buildSrc project

You can put the source for the plugin in the `rootProjectDir/buildSrc/src/main/groovy` directory. Gradle will take care of compiling and testing the plugin and making it available on the classpath of the build script. The plugin is visible to every build script used by the build. However, it is not visible outside the build, and so you cannot reuse the plugin outside the build it is defined in.

See Chapter 43, *Organizing Build Logic* for more details about the `buildSrc` project.

Standalone project

You can create a separate project for your plugin. This project produces and publishes a JAR which you can then use in multiple builds and share with others. Generally, this JAR might include some custom plugins, or bundle several related task classes into a single library. Or some combination of the two.

In our examples, we will start with the plugin in the build script, to keep things simple. Then we will look at creating a standalone project.

41.2. Writing a simple plugin

To create a custom plugin, you need to write an implementation of `Plugin`. Gradle instantiates the plugin and calls the plugin instance's `Plugin.apply(T)` 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 41.1. A custom plugin

build.gradle

```
apply plugin: GreetingPlugin

class GreetingPlugin implements Plugin<Project> {
    void apply(Project project) {
        project.task('hello') {
            doLast {
                println "Hello from the GreetingPlugin"
            }
        }
    }
}
```

Output of **gradle -q hello**

```
> gradle -q hello
Hello from the GreetingPlugin
```

One thing to note is that a new instance of a given plugin is created for each project it is applied to. Also note that the `Plugin` class is a generic type. This example has it receiving the `Project` type as a type parameter. It's possible to write unusual custom plugins that take different type parameters, but this will be unlikely (until someone figures out more creative things to do here).

41.3. Getting input from the build

Most plugins need to obtain some configuration from the build script. One method for doing this is to use *extension objects*. The Gradle `Project` has an associated `ExtensionContainer` object that helps keep track of all the settings and properties being passed to plugins. You can capture user input by telling the extension container about your plugin. To capture input, simply add a Java Bean compliant class into the extension container's list of extensions. Groovy is a good language choice for a plugin because plain old Groovy objects contain all the getter and setter methods that a Java Bean requires.

Let's add a simple extension object to the project. Here we add a `greeting` extension object to the project, which allows you to configure the greeting.

Example 41.2. A custom plugin extension

build.gradle

```
apply plugin: GreetingPlugin

greeting.message = 'Hi from Gradle'

class GreetingPlugin implements Plugin<Project> {
    void apply(Project project) {
        // Add the 'greeting' extension object
        project.extensions.create("greeting", GreetingPluginExtension)
        // Add a task that uses the configuration
        project.task('hello') {
            doLast {
                println project.greeting.message
            }
        }
    }
}

class GreetingPluginExtension {
    def String message = 'Hello from GreetingPlugin'
}
```

Output of **gradle -q hello**

```
> gradle -q hello
Hi from Gradle
```

In this example, `GreetingPluginExtension` is a plain old Groovy object with a field called `message`. The extension object is added to the plugin list with the name `greeting`. This object then becomes available as a project property with the same name as the extension object.

Oftentimes, you have several related properties you need to specify on a single plugin. Gradle adds a configuration closure block for each extension object, so you can group settings together. The following example shows you how this works.

Example 41.3. A custom plugin with configuration closure

build.gradle

```
apply plugin: GreetingPlugin

greeting {
    message = 'Hi'
    greeter = 'Gradle'
}

class GreetingPlugin implements Plugin<Project> {
    void apply(Project project) {
        project.extensions.create("greeting", GreetingPluginExtension)
        project.task('hello') {
            doLast {
                println "${project.greeting.message} from ${project.greeting.greeter}"
            }
        }
    }
}

class GreetingPluginExtension {
    String message
    String greeter
}
```

Output of `gradle -q hello`

```
> gradle -q hello
Hi from Gradle
```

In this example, several settings can be grouped together within the `greeting` closure. The name of the closure block in the build script (`greeting`) needs to match the extension object name. Then, when the closure is executed, the fields on the extension object will be mapped to the variables within the closure based on the standard Groovy closure delegate feature.

41.4. Working with files in custom tasks and plugins

When developing custom tasks and plugins, it's a good idea to be very flexible when accepting input configuration for file locations. To do this, you can leverage the `Project.file(java.lang.Object)` method to resolve values to files as late as possible.

Example 41.4. Evaluating file properties lazily

build.gradle

```
class GreetingToFileTask extends DefaultTask {

    def destination

    File getDestination() {
        project.file(destination)
    }

    @TaskAction
    def greet() {
        def file = getDestination()
        file.parentFile.mkdirs()
        file.write "Hello!"
    }
}

task greet(type: GreetingToFileTask) {
    destination = { project.greetingFile }
}

task sayGreeting(dependsOn: greet) {
    doLast {
        println file(greetingFile).text
    }
}

ext.greetingFile = "$buildDir/hello.txt"
```

Output of **gradle -q sayGreeting**

```
> gradle -q sayGreeting
Hello!
```

In this example, we configure the `greet` task `destination` property as a closure, which is evaluated with the `Project.file(java.lang.Object)` method to turn the return value of the closure into a file object at the last minute. You will notice that in the example above we specify the `greetingFile` property value after we have configured to use it for the task. This kind of lazy evaluation is a key benefit of accepting any value when setting a file property, then resolving that value when reading the property.

41.5. A standalone project

Now we will move our plugin to a standalone project, so we can publish it and share it with others. This project is simply a Groovy project that produces a JAR containing the plugin classes. Here is a simple build script for the project. It applies the Groovy plugin, and adds the Gradle API as a compile-time dependency.

Example 41.5. A build for a custom plugin

build.gradle

```
apply plugin: 'groovy'

dependencies {
    compile gradleApi()
    compile localGroovy()
}
```

Note: The code for this example can be found at `samples/customPlugin/plugin` in the ‘-all’ distribution of Gradle.

So how does Gradle find the Plugin implementation? The answer is you need to provide a properties file in the jar's `META-INF/gradle-plugins` directory that matches the id of your plugin.

Example 41.6. Wiring for a custom plugin

`src/main/resources/META-INF/gradle-plugins/org.samples.greeting.properties`

```
implementation-class=org.gradle.GreetingPlugin
```

Notice that the properties filename matches the plugin id and is placed in the resources folder, and that the `implementation-class` property identifies the Plugin implementation class.

41.5.1. Creating a plugin id

Plugin ids are fully qualified in a manner similar to Java packages (i.e. a reverse domain name). This helps to avoid collisions and provides a way to group plugins with similar ownership.

Your plugin id should be a combination of components that reflect namespace (a reasonable pointer to you or your organization) and the name of the plugin it provides. For example if you had a Github account named “foo” and your plugin was named “bar”, a suitable plugin id might be `com.github.foo.bar`. Similarly, if the plugin was developed at the baz organization, the plugin id might be `org.baz.bar`.

Plugin ids should conform to the following:

- May contain any alphanumeric character, '.', and '-'.
- Must contain at least one '.' character separating the namespace from the name of the plugin.
- Conventionally use a lowercase reverse domain name convention for the namespace.
- Conventionally use only lowercase characters in the name.
- `org.gradle` and `com.gradleware` namespaces may not be used.
- Cannot start or end with a '.' character.
- Cannot contain consecutive '.' characters (i.e. '..').

Although there are conventional similarities between plugin ids and package names, package names are generally more detailed than is necessary for a plugin id. For instance, it might seem reasonable to add

“gradle” as a component of your plugin id, but since plugin ids are only used for Gradle plugins, this would be superfluous. Generally, a namespace that identifies ownership and a name are all that are needed for a good plugin id.

41.5.2. Publishing your plugin

If you are publishing your plugin internally for use within your organization, you can publish it like any other code artifact. See the [ivy](#) and [maven](#) chapters on publishing artifacts.

If you are interested in publishing your plugin to be used by the wider Gradle community, you can publish it to the Gradle plugin portal. This site provides the ability to search for and gather information about plugins contributed by the Gradle community. See the [instructions](#) here on how to make your plugin available on this site.

41.5.3. Using your plugin in another project

To use a plugin in a build script, you need to add the plugin classes to the build script's classpath. To do this, you use a “`buildscript { }`” block, as described in the section called “Applying plugins with the buildscript block”. The following example shows how you might do this when the JAR containing the plugin has been published to a local repository:

Example 41.7. Using a custom plugin in another project

build.gradle

```
buildscript {
    repositories {
        maven {
            url uri('../repo')
        }
    }
    dependencies {
        classpath group: 'org.gradle', name: 'customPlugin',
            version: '1.0-SNAPSHOT'
    }
}
apply plugin: 'org.samples.greeting'
```

Alternatively, if your plugin is published to the plugin portal, you can use the incubating plugins DSL (see [Section 27.5.2](#), “Applying plugins with the plugins DSL”) to apply the plugin:

Example 41.8. Applying a community plugin with the plugins DSL

build.gradle

```
plugins {
    id "com.jfrog.bintray" version "0.4.1"
}
```

41.5.4. Writing tests for your plugin

You can use the `ProjectBuilder` class to create `Project` instances to use when you test your plugin implementation.

Example 41.9. Testing a custom plugin

src/test/groovy/org/gradle/GreetingPluginTest.groovy

```
class GreetingPluginTest {
    @Test
    public void greeterPluginAddsGreetingTaskToProject() {
        Project project = ProjectBuilder.builder().build()
        project.pluginManager.apply 'org.samples.greeting'

        assertTrue(project.tasks.hello instanceof GreetingTask)
    }
}
```

41.5.5. Using the Java Gradle Plugin development plugin

You can use the incubating Java Gradle Plugin development plugin to eliminate some of the boilerplate declarations in your build script and provide some basic validations of plugin metadata. This plugin will automatically apply the Java plugin, add the `gradleApi()` dependency to the compile configuration, and perform plugin metadata validations as part of the `jar` task execution.

Example 41.10. Using the Java Gradle Plugin Development plugin

build.gradle

```
plugins {
    id "java-gradle-plugin"
}
```

When publishing plugins to custom plugin repositories using the ivy or maven publish plugins, the Java Gradle Plugin will also generate plugin marker artifacts named based on the plugin id which depend on the plugin's implementation artifact.

41.6. Maintaining multiple domain objects

Gradle provides some utility classes for maintaining collections of objects, which work well with the Gradle build language.

Example 41.11. Managing domain objects

build.gradle

```
apply plugin: DocumentationPlugin

books {
    quickStart {
        sourceFile = file('src/docs/quick-start')
    }
    userGuide {
    }
    developerGuide {
    }
}

task books {
    doLast {
        books.each { book ->
            println "$book.name -> $book.sourceFile"
        }
    }
}

class DocumentationPlugin implements Plugin<Project> {
    void apply(Project project) {
        def books = project.container(Book)
        books.all {
            sourceFile = project.file("src/docs/$name")
        }
        project.extensions.books = books
    }
}

class Book {
    final String name
    File sourceFile

    Book(String name) {
        this.name = name
    }
}
```

Output of `gradle -q books`

```
> gradle -q books
developerGuide -> /home/user/gradle/samples/userguide/organizeBuildLogic/customPluginWith
quickStart -> /home/user/gradle/samples/userguide/organizeBuildLogic/customPluginWith
userGuide -> /home/user/gradle/samples/userguide/organizeBuildLogic/customPluginWith
```

The `Project.container(java.lang.Class)` methods create instances of `NamedDomainObjectContainer`, that have many useful methods for managing and configuring the objects. In order to use a type with any of the `project.container` methods, it MUST expose a property named “name” as the unique, and constant, name for the object. The `project.container(Class)`

variant of the container method creates new instances by attempting to invoke the constructor of the class that takes a single string argument, which is the desired name of the object. See the above link for `project.con` method variants that allow custom instantiation strategies.

The Java Gradle Plugin Development Plugin

The Java Gradle plugin development plugin is currently incubating. Please be aware that the DSL and other configuration may change in later Gradle versions.

The Java Gradle Plugin development plugin can be used to assist in the development of Gradle plugins. It automatically applies the Java plugin, adds the `gradleApi()` dependency to the compile configuration and performs validation of plugin metadata during `jar` task execution.

The plugin also integrates with TestKit, a library that aids in writing and executing functional tests for plugin code. It automatically adds the `gradleTestKit()` dependency to the test compile configuration and generates a plugin classpath manifest file consumed by a `GradleRunner` instance if found. Please refer to the section called “Automatic injection with the Java Gradle Plugin Development plugin” for more on its usage, configuration options and samples.

42.1. Usage

To use the Java Gradle Plugin Development plugin, include the following in your build script:

Example 42.1. Using the Java Gradle Plugin Development plugin

build.gradle

```
plugins {  
    id "java-gradle-plugin"  
}
```

Applying the plugin automatically applies the Java plugin and adds the `gradleApi()` dependency to the compile configuration. It also adds some validations to the build.

The following validations are performed:

- There is a plugin descriptor defined for the plugin.
- The plugin descriptor contains an `implementation-class` property.
- The `implementation-class` property references a valid class file in the jar.
- Each property getter or the corresponding field must be annotated with a property annotation like `@InputFile`

and `@OutputDirectory`. Properties that don't participate in up-to-date checks should be annotated with `@Internal`.

Any failed validations will result in a warning message.

For each plugin you are developing, add an entry to the `gradlePlugin {}` script block:

Example 42.2. Using the `gradlePlugin {}` block.

build.gradle

```
gradlePlugin {
    plugins {
        simplePlugin {
            id = "org.gradle.sample.simple-plugin"
            implementationClass = "org.gradle.sample.SimplePlugin"
        }
    }
}
```

The `gradlePlugin {}` block defines the plugins being built by the project including the `id` and `implementationClass` of the plugin. From this data about the plugins being developed, Gradle can automatically:

- Generate the plugin descriptor in the `jar` file's `META-INF` directory.
- Configure the Maven or Ivy publishing plugins to publish a Plugin Marker Artifact for each plugin.

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 then you likely should implement your logic with classes to encapsulate your logic. ^[21] Gradle makes this very easy. Just drop your classes in a certain directory and Gradle automatically compiles them and puts them in the classpath of your build script.

Here is a summary of the ways you can organise your build logic:

- **POGOs.** You can declare and use plain old Groovy objects (POGOs) directly in your build script. The build script is written in Groovy, after all, and Groovy provides you with lots of excellent ways to organize code.
- **Inherited properties and methods.** In a multi-project build, sub-projects inherit the properties and methods of their parent project.
- **Configuration injection.** In a multi-project build, a project (usually the root project) can inject properties and methods into another project.
- **buildSrc project.** Drop the source for your build classes into a certain directory and Gradle automatically compiles them and includes them in the classpath of your build script.
- **Shared scripts.** Define common configuration in an external build, and apply the script to multiple projects, possibly across different builds.
- **Custom tasks.** Put your build logic into a custom task, and reuse that task in multiple places.
- **Custom plugins.** Put your build logic into a custom plugin, and apply that plugin to multiple projects. The plugin must be in the classpath of your build script. You can achieve this either by using `build source` or by adding an external library that contains the plugin.
- **Execute an external build.** Execute another Gradle build from the current build.
- **External libraries.** Use external libraries directly in your build file.

43.1. Inherited properties and methods

Any method or property defined in a project build script is also visible to all the sub-projects. You can use this to define common configurations, and to extract build logic into methods which can be reused by the sub-projects.

Example 43.1. Using inherited properties and methods

build.gradle

```
// Define an extra property
ext.srcDirName = 'src/java'

// Define a method
def getSrcDir(project) {
    return project.file(srcDirName)
}
```

child/build.gradle

```
task show {
    doLast {
        // Use inherited property
        println 'srcDirName: ' + srcDirName

        // Use inherited method
        File srcDir = getSrcDir(project)
        println 'srcDir: ' + rootProject.relativePath(srcDir)
    }
}
```

Output of **gradle -q show**

```
> gradle -q show
srcDirName: src/java
srcDir: child/src/java
```

43.2. Injected configuration

You can use the configuration injection technique discussed in Section 26.1, “Cross project configuration” and Section 26.2, “Subproject configuration” to inject properties and methods into various projects. This is generally a better option than inheritance, for a number of reasons: The injection is explicit in the build script, You can inject different logic into different projects, And you can inject any kind of configuration such as repositories, plug-ins, tasks, and so on. The following sample shows how this works.

Example 43.2. Using injected properties and methods

build.gradle

```
subprojects {
    // Define a new property
    ext.srcDirName = 'src/java'

    // Define a method using a closure as the method body
    ext.srcDir = { file(srcDirName) }

    // Define a task
    task show {
        doLast {
            println 'project: ' + project.path
            println 'srcDirName: ' + srcDirName
            File srcDir = srcDir()
            println 'srcDir: ' + rootProject.relativePath(srcDir)
        }
    }
}

// Inject special case configuration into a particular project
project(':child2') {
    ext.srcDirName = "$srcDirName/legacy"
}
```

child1/build.gradle

```
// Use injected property and method. Here, we override the injected value
srcDirName = 'java'
def dir = srcDir()
```

Output of `gradle -q show`

```
> gradle -q show
project: :child1
srcDirName: java
srcDir: child1/java
project: :child2
srcDirName: src/java/legacy
srcDir: child2/src/java/legacy
```

43.3. Configuring the project using an external build script

You can configure the current project using an external build script. All of the Gradle build language is available in the external script. You can even apply other scripts from the external script.

Example 43.3. Configuring the project using an external build script

build.gradle

```
apply from: 'other.gradle'
```

other.gradle

```
println "configuring $project"
task hello {
    doLast {
        println 'hello from other script'
    }
}
```

Output of **gradle -q hello**

```
> gradle -q hello
configuring root project 'configureProjectUsingScript'
hello from other script
```

43.4. Build sources in the buildSrc project

When you run Gradle, it checks for the existence of a directory called `buildSrc`. 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. This can be a good place to add your custom tasks and plugins.

For multi-project builds there can be only one `buildSrc` directory, which has to be in the root project directory.

Listed below is the default build script that Gradle applies to the `buildSrc` project:

Figure 43.1. Default buildSrc build script

```
apply plugin: 'groovy'
dependencies {
    compile gradleApi()
    compile localGroovy()
}
```

This means that you can just put your build source code in this directory and stick to the layout convention for a Java/Groovy project (see Table 47.4, “Java plugin - default project layout”).

If you need more flexibility, you can provide your own `build.gradle`. Gradle applies the default build script regardless of whether there is one specified. This means you only need to declare the extra things you need. Below is an example. Notice that this example does not need to declare a dependency on the Gradle API, as this is done by the default build script:

Example 43.4. Custom buildSrc build script

buildSrc/build.gradle

```
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    testCompile 'junit:junit:4.12'  
}
```

The buildSrc project can be a multi-project build, just like any other regular multi-project build. However, all of the projects that should be on the classpath of the actual build must be runtime dependencies of the root project in buildSrc. You can do this by adding this to the configuration of each project you wish to export:

Example 43.5. Adding subprojects to the root buildSrc project

buildSrc/build.gradle

```
rootProject.dependencies {  
    runtime project(path)  
}
```

Note: The code for this example can be found at `samples/multiProjectBuildSrc` in the ‘-all’ distribution of Gradle.

43.5. Running another Gradle build from a build

You can use the GradleBuild task. You can use either of the `dir` or `buildFile` properties to specify which build to execute, and the `tasks` property to specify which tasks to execute.

Example 43.6. Running another build from a build

build.gradle

```
task build(type: GradleBuild) {
    buildFile = 'other.gradle'
    tasks = ['hello']
}
```

other.gradle

```
task hello {
    doLast {
        println "hello from the other build."
    }
}
```

Output of **gradle -q build**

```
> gradle -q build
hello from the other build.
```

43.6. 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 43.7. 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 25.4, “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 43.8. 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 {
    doLast {
        def byte[] encodedString = new Base64().encode('hello world\n'.getBytes())
        println new String(encodedString)
    }
}
```

Output of **gradle -q encode**

```
> gradle -q encode
aGVsbG8gd29ybGQK
```

For multi-project builds, the dependencies declared with a project's `buildscript()` method are available to the build scripts of all its sub-projects.

Build script dependencies may be Gradle plugins. Please consult Chapter 27, *Gradle Plugins* for more information on Gradle plugins.

Every project automatically has a `buildEnvironment` task of type `BuildEnvironmentReportTask` that can be invoked to report on the resolution of the build script dependencies.

43.7. 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. ^[22]

Example 43.9. Ant optional dependencies

build.gradle

```
configurations {
    ftpAntTask
}

dependencies {
    ftpAntTask("org.apache.ant:ant-commons-net:1.9.6") {
        module("commons-net:commons-net:1.4.1") {
            dependencies "oro:oro:2.0.8:jar"
        }
    }
}

task ftp {
    doLast {
        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.c",
                fileset(dir: "htdocs/manual"))
        }
    }
}
```

This is also a good example for the usage of client modules. The POM file in Maven Central for the ant-commons-net task does not provide the right information for this use case.

43.8. 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.

[21] Which might range from a single class to something very complex.

[22] In fact, we think this is a better solution. Only if your buildscript and Ant's optional task need the *same* library would you have to define it twice. In such a case it would be nice if Ant's optional task would automatically pick up the classpath defined in the “gradle.settings” file.

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.

Note that this is completely different from the “init” task provided by the “build-init” incubating plugin (see Chapter 17, *Build Init Plugin*).

44.1. 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 enterprise-wide configuration, such as where to find custom plugins.
- 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 that is required by 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.
- Register build loggers. You might wish to customize how Gradle logs the events that it generates.

One main limitation of init scripts is that they cannot access classes in the `buildSrc` project (see Section 43.4, “Build sources in the `buildSrc` project” for details of this feature).

44.2. Using an init script

There are several ways to use an init script:

- Specify a 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.
- Put a file called `init.gradle` in the `USER_HOME/.gradle/` directory.
- Put a file that ends with `.gradle` in the `USER_HOME/.gradle/init.d/` directory.
- Put a file that ends with `.gradle` in the `GRADLE_HOME/init.d/` directory, in the Gradle distribution. This allows you to package up a custom Gradle distribution containing some custom build logic and plugins. You can combine this with the Gradle wrapper as a way to make custom logic available to all builds in your enterprise.

If more than one init script is found they will all be executed, in the order specified above. Scripts in a given directory are executed in alphabetical order. This allows, for example, a tool to specify an init script on the command line and the user to put one in their home directory for defining the environment and both scripts will run when Gradle is executed.

44.3. Writing an init script

Similar to a Gradle build script, an init script is a Groovy script. Each init script has a Gradle instance associated with it. Any property reference and method call in the init script will delegate to this Gradle instance.

Each init script also implements the `Script` interface.

44.3.1. Configuring projects from an init script

You can use an init script to configure the projects in the build. This works in a similar way to configuring projects in a multi-project build. The following sample shows how to perform extra configuration from an init script *before* the projects are evaluated. This sample uses this feature to configure an extra repository to be used only for certain environments.

Example 44.1. Using init script to perform extra configuration before projects are evaluated

build.gradle

```
repositories {
    mavenCentral()
}

task showRepos {
    doLast {
        println "All repos:"
        println repositories.collect { it.name }
    }
}
```

init.gradle

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

Output of **gradle --init-script init.gradle -q showRepos**

```
> gradle --init-script init.gradle -q showRepos
All repos:
[MavenLocal, MavenRepo]
```

44.4. External dependencies for the init script

In Section 43.6, “External dependencies for the build script” it was explained how to add external dependencies to a build script. Init scripts can also declare dependencies. You do this with the `initscript()` method, passing in a closure which declares the init script classpath.

Example 44.2. 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 25.4, “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 44.3. 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
```


44.5. Init script plugins

Similar to a Gradle build script or a Gradle settings file, plugins can be applied on init scripts.

Example 44.4. Using plugins in init scripts

init.gradle

```
apply plugin:EnterpriseRepositoryPlugin

class EnterpriseRepositoryPlugin implements Plugin<Gradle> {

    private static String ENTERPRISE_REPOSITORY_URL = "https://repo.gradle.org/gradle"

    void apply(Gradle gradle) {
        // ONLY USE ENTERPRISE REPO FOR DEPENDENCIES
        gradle.allprojects{ project ->
            project.repositories {

                // Remove all repositories not pointing to the enterprise repository
                all { ArtifactRepository repo ->
                    if (!(repo instanceof MavenArtifactRepository) ||
                        repo.url.toString() != ENTERPRISE_REPOSITORY_URL) {
                        project.logger.lifecycle "Repository ${repo.url} removed. On
remove repo
                    }
                }

                // add the enterprise repository
                maven {
                    name "STANDARD_ENTERPRISE_REPO"
                    url ENTERPRISE_REPOSITORY_URL
                }
            }
        }
    }
}
```

build.gradle

```
repositories{
    mavenCentral()
}

task showRepositories {
    doLast {
        repositories.each {
            println "repository: ${it.name} ('${it.url}')"
        }
    }
}
```

Output of `gradle -q -I init.gradle showRepositories`

```
> gradle -q -I init.gradle showRepositories
repository: STANDARD_ENTERPRISE_REPO ('https://repo.gradle.org/gradle/repo')
```

The plugin in the init script ensures that only a specified repository is used when running the build.

When applying plugins within the init script, Gradle instantiates the plugin and calls the plugin instance's `Plugin.apply(T)` method. The `gradle` object is passed as a parameter, which can be used to configure all aspects of a build. Of course, the applied plugin can be resolved as an external dependency as described in Section 44.4, “External dependencies for the init script”

The Gradle TestKit

The Gradle TestKit is currently incubating. Please be aware that its API and other characteristics may change in later Gradle versions.

The Gradle TestKit (a.k.a. just TestKit) is a library that aids in testing Gradle plugins and build logic generally. At this time, it is focused on *functional* testing. That is, testing build logic by exercising it as part of a programmatically executed build. Over time, the TestKit will likely expand to facilitate other kinds of tests.

45.1. Usage

To use the TestKit, include the following in your plugin's build:

Example 45.1. Declaring the TestKit dependency

build.gradle

```
dependencies {  
    testCompile gradleTestKit()  
}
```

The `gradleTestKit()` encompasses the classes of the TestKit, as well as the Gradle Tooling API client. It does not include a version of JUnit, TestNG, or any other test execution framework. Such a dependency must be explicitly declared.

Example 45.2. Declaring the JUnit dependency

build.gradle

```
dependencies {  
    testCompile 'junit:junit:4.12'  
}
```

45.2. Functional testing with the Gradle runner

The `GradleRunner` facilitates programmatically executing Gradle builds, and inspecting the result.

A contrived build can be created (e.g. programmatically, or from a template) that exercises the “logic under

test”. The build can then be executed, potentially in a variety of ways (e.g. different combinations of tasks and arguments). The correctness of the logic can then be verified by asserting the following, potentially in combination:

- The build's output;
- The build's logging (i.e. console output);
- The set of tasks executed by the build and their results (e.g. FAILED, UP-TO-DATE etc.).

After creating and configuring a runner instance, the build can be executed via the `GradleRunner.build()` or `GradleRunner.buildAndFail()` methods depending on the anticipated outcome.

The following demonstrates the usage of Gradle runner in a Java JUnit test:

Example 45.3. Using GradleRunner with JUnit

BuildLogicFunctionalTest.java

```
import org.gradle.testkit.runner.BuildResult;
import org.gradle.testkit.runner.GradleRunner;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.TemporaryFolder;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Collections;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

import static org.gradle.testkit.runner.TaskOutcome.*;

public class BuildLogicFunctionalTest {
    @Rule public final TemporaryFolder testProjectDir = new TemporaryFolder();
    private File buildFile;

    @Before
    public void setup() throws IOException {
        buildFile = testProjectDir.newFile("build.gradle");
    }

    @Test
    public void testHelloWorldTask() throws IOException {
        String buildFileContent = "task helloWorld {" +
                                   "    doLast {" +
                                   "        println 'Hello world!'" +
                                   "    }" +
                                   "}";

        writeFile(buildFile, buildFileContent);

        BuildResult result = GradleRunner.create()
            .withProjectDir(testProjectDir.getRoot())
            .withArguments("helloWorld")
            .build();

        assertTrue(result.getOutput().contains("Hello world!"));
        assertEquals(result.task(":helloWorld").getOutcome(), SUCCESS);
    }

    private void writeFile(File destination, String content) throws IOException {
        BufferedWriter output = null;
        try {
            output = new BufferedWriter(new FileWriter(destination));
            output.write(content);
        } finally {
            if (output != null) {
                output.close();
            }
        }
    }
}
```

Any test execution framework can be used.

As Gradle build scripts are written in the Groovy programming language, and as many plugins are implemented in Groovy, it is often a productive choice to write Gradle functional tests in Groovy. Furthermore, it is recommended to use the (Groovy based) Spock test execution framework as it offers many compelling features over the use of JUnit.

The following demonstrates the usage of Gradle runner in a Groovy Spock test:

Example 45.4. Using GradleRunner with Spock

BuildLogicFunctionalTest.groovy

```
import org.gradle.testkit.runner.GradleRunner
import static org.gradle.testkit.runner.TaskOutcome.*
import org.junit.Rule
import org.junit.rules.TemporaryFolder
import spock.lang.Specification

class BuildLogicFunctionalTest extends Specification {
    @Rule final TemporaryFolder testProjectDir = new TemporaryFolder()
    File buildFile

    def setup() {
        buildFile = testProjectDir.newFile('build.gradle')
    }

    def "hello world task prints hello world"() {
        given:
        buildFile << """
            task helloWorld {
                doLast {
                    println 'Hello world!'
                }
            }
        """

        when:
        def result = GradleRunner.create()
            .withProjectDir(testProjectDir.root)
            .withArguments('helloWorld')
            .build()

        then:
        result.output.contains('Hello world!')
        result.task(":helloWorld").outcome == SUCCESS
    }
}
```

It is a common practice to implement any custom build logic (like plugins and task types) that is more complex in nature as external classes in a standalone project. The main driver behind this approach is bundle the compiled code into a JAR file, publish it to a binary repository and reuse it across various projects.

45.2.1. Getting the plugin-under-test into the test build

The GradleRunner uses the Tooling API to execute builds. An implication of this is that the builds are executed in a separate process (i.e. not the same process executing the tests). Therefore, the test build does not share the same classpath or classloaders as the test process and the code under test is not implicitly available to the test build.

Starting with version 2.13, Gradle provides a conventional mechanism to inject the code under test into the test build.

For earlier versions of Gradle (before 2.13), it is possible to manually make the code under test available via some extra configuration. The following example demonstrates having the build generate a file containing the implementation classpath of the code under test, and making it available at test runtime.

Example 45.5. Making the code under test classpath available to the tests

build.gradle

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

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

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

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

Note: The code for this example can be found at `samples/testKit/gradleRunner/manualClasspath` in the ‘-all’ distribution of Gradle.

The tests can then read this value, and inject the classpath into the test build by using the method `GradleRunner.withPluginClasspath(java.lang.Iterable)`. This classpath is then available to use to locate plugins in a test build via the plugins DSL (see Chapter 27, *Gradle Plugins*). Applying plugins with the plugins DSL requires the definition of a plugin identifier. The following is an example (in Groovy) of doing this from within a Spock Framework `setup()` method, which is analogous to a JUnit `@Before` method.

This approach works well when executing the functional tests as part of the Gradle build. When executing the functional tests from an IDE, there are extra considerations. Namely, the classpath manifest file points to the class files etc. generated by Gradle and not the IDE. This means that after making a change to the source

of the code under test, the source must be recompiled by Gradle. Similarly, if the effective classpath of the code under test changes, the manifest must be regenerated. In either case, executing the `testClasses` task of the build will ensure that things are up to date.

Working with Gradle versions prior to 2.8

The `GradleRunner.withPluginClasspath(java.lang.Iterable)` method will not work when executing the build with a Gradle version earlier than 2.8 (see: Section 45.2.3, “The Gradle version used to test”), as this feature is not supported on such Gradle versions.

Instead, the code must be injected via the build script itself. The following sample demonstrates how this can be done.

Example 45.6. Injecting the code under test classes into test builds

src/test/groovy/org/gradle/sample/BuildLogicFunctionalTest.groovy

```
List<File> pluginClasspath

def setup() {
    buildFile = testProjectDir.newFile('build.gradle')

    def pluginClasspathResource = getClass().classLoader.findResource("plugin-classp
    if (pluginClasspathResource == null) {
        throw new IllegalStateException("Did not find plugin classpath resource, run
    }

    pluginClasspath = pluginClasspathResource.readLines().collect { new File(it)
}

def "hello world task prints hello world"() {
    given:
    buildFile << """
        plugins {
            id 'org.gradle.sample.helloworld'
        }
    """

    when:
    def result = GradleRunner.create()
        .withProjectDir(testProjectDir.root)
        .withArguments('helloWorld')
        .withPluginClasspath(pluginClasspath)
        .build()

    then:
    result.output.contains('Hello world!')
    result.task(":helloWorld").outcome == SUCCESS
}
```

Note: The code for this example can be found at `samples/testKit/gradleRunner/manualClass` in the ‘-all’ distribution of Gradle.

src/test/groovy/org/gradle/sample/BuildLogicFunctionalTest.groovy


```

List<File> pluginClasspath

def setup() {
    buildFile = testProjectDir.newFile('build.gradle')

    def pluginClasspathResource = getClass().classLoader.findResource("plugin-classp
    if (pluginClasspathResource == null) {
        throw new IllegalStateException("Did not find plugin classpath resource, run
    }

    pluginClasspath = pluginClasspathResource.readLines().collect { new File(it)
}

def "hello world task prints hello world with pre Gradle 2.8"() {
    given:
    def classpathString = pluginClasspath
        .collect { it.absolutePath.replace('\\', '\\\\') } // escape backslashes in
        .collect { "'$it'" }
        .join(", ")

    buildFile << """
        buildscript {
            dependencies {
                classpath files($classpathString)
            }
        }
        apply plugin: "org.gradle.sample.helloworld"
    """

    when:
    def result = GradleRunner.create()
        .withProjectDir(testProjectDir.root)
        .withArguments('helloWorld')
        .withGradleVersion("2.7")
        .build()

    then:
    result.output.contains('Hello world!')
    result.task(":helloWorld").outcome == SUCCESS
}

```

Note: The code for this example can be found at `samples/testKit/gradleRunner/manualClasspathInjection` in the ‘-all’ distribution of Gradle.

Automatic injection with the Java Gradle Plugin Development plugin

The Java Gradle Plugin development plugin can be used to assist in the development of Gradle plugins. Starting with Gradle version 2.13, the plugin provides a direct integration with TestKit. When applied to a project, the plugin automatically adds the `gradleTestKit()` dependency to the test compile configuration. Furthermore, it automatically generates the classpath for the code under test and injects it via `GradleRunner.withPluginClasspath()` for any `GradleRunner` instance created by the user. If the target Gradle version is prior to 2.8, automatic plugin classpath injection is not performed.

The plugin uses the following conventions for applying the TestKit dependency and injecting the classpath:

- Source set containing code under test: `sourceSets.main`
- Source set used for injecting the plugin classpath: `sourceSets.test`

Any of these conventions can be reconfigured with the help of the class `GradlePluginDevelopmentExtension`.

The following Groovy-based sample demonstrates how to automatically inject the plugin classpath by using the standard conventions applied by the Java Gradle Plugin Development plugin.

Example 45.7. Using the Java Gradle Development plugin for generating the plugin metadata

build.gradle

```
apply plugin: 'groovy'
apply plugin: 'java-gradle-plugin'

dependencies {
    testCompile('org.spockframework:spock-core:1.0-groovy-2.4') {
        exclude module: 'groovy-all'
    }
}
```

Note: The code for this example can be found at `samples/testKit/gradleRunner/automaticClasspath` in the ‘-all’ distribution of Gradle.

Example 45.8. Automatically injecting the code under test classes into test builds

src/test/groovy/org/gradle/sample/BuildLogicFunctionalTest.groovy

```
def "hello world task prints hello world"() {
    given:
    buildFile << """
        plugins {
            id 'org.gradle.sample.helloworld'
        }
    """

    when:
    def result = GradleRunner.create()
        .withProjectDir(testProjectDir.root)
        .withArguments('helloWorld')
        .withPluginClasspath()
        .build()

    then:
    result.output.contains('Hello world!')
    result.task(":helloWorld").outcome == SUCCESS
}
```

Note: The code for this example can be found at `samples/testKit/gradleRunner/automaticClasspath` in the ‘-all’ distribution of Gradle.

The following build script demonstrates how to reconfigure the conventions provided by the Java Gradle Plugin Development plugin for a project that uses a custom Test source set.

Example 45.9. Reconfiguring the classpath generation conventions of the Java Gradle Development plugin

build.gradle

```
apply plugin: 'groovy'
apply plugin: 'java-gradle-plugin'

sourceSets {
    functionalTest {
        groovy {
            srcDir file('src/functionalTest/groovy')
        }
        resources {
            srcDir file('src/functionalTest/resources')
        }
        compileClasspath += sourceSets.main.output + configurations.testRuntime
        runtimeClasspath += output + compileClasspath
    }
}

task functionalTest(type: Test) {
    testClassesDir = sourceSets.functionalTest.output.classesDir
    classpath = sourceSets.functionalTest.runtimeClasspath
}

check.dependsOn functionalTest

gradlePlugin {
    testSourceSets sourceSets.functionalTest
}

dependencies {
    functionalTestCompile('org.spockframework:spock-core:1.0-groovy-2.4') {
        exclude module: 'groovy-all'
    }
}
```

Note: The code for this example can be found at `samples/testKit/gradleRunner/automaticClasspath` in the ‘-all’ distribution of Gradle.

45.2.2. Controlling the build environment

The runner executes the test builds in an isolated environment by specifying a dedicated "working directory" in a directory inside the JVM's temp directory (i.e. the location specified by the `java.io.tmpdir` system property, typically `/tmp`). Any configuration in the default Gradle user home directory (e.g. `~/ .gradle/gradle`) is not used for test execution. The TestKit does not expose a mechanism for fine grained control of environment variables etc. Future versions of the TestKit will provide improved configuration options.

The TestKit uses dedicated daemon processes that are automatically shut down after test execution.

45.2.3. The Gradle version used to test

The Gradle runner requires a Gradle distribution in order to execute the build. The `TestKit` does not depend on all of Gradle's implementation.

By default, the runner will attempt to find a Gradle distribution based on where the `GradleRunner` class was loaded from. That is, it is expected that the class was loaded from a Gradle distribution, as is the case when using the `gradleTestKit()` dependency declaration.

When using the runner as part of tests *being executed by Gradle* (e.g. executing the `test` task of a plugin project), the same distribution used to execute the tests will be used by the runner. When using the runner as part of tests *being executed by an IDE*, the same distribution of Gradle that was used when importing the project will be used. This means that the plugin will effectively be tested with the same version of Gradle that it is being built with.

Alternatively, a different and specific version of Gradle to use can be specified by the any of the following `GradleRunner` methods:

- `GradleRunner.withGradleVersion(java.lang.String)`
- `GradleRunner.withGradleInstallation(java.io.File)`
- `GradleRunner.withGradleDistribution(java.net.URI)`

This can potentially be used to test build logic across Gradle versions. The following demonstrates a cross-version compatibility test written as Groovy Spock test:

Example 45.10. Specifying a Gradle version for test execution

BuildLogicFunctionalTest.groovy

```
import org.gradle.testkit.runner.GradleRunner
import static org.gradle.testkit.runner.TaskOutcome.*
import org.junit.Rule
import org.junit.rules.TemporaryFolder
import spock.lang.Specification
import spock.lang.Unroll

class BuildLogicFunctionalTest extends Specification {
    @Rule final TemporaryFolder testProjectDir = new TemporaryFolder()
    File buildFile

    def setup() {
        buildFile = testProjectDir.newFile('build.gradle')
    }

    @Unroll
    def "can execute hello world task with Gradle version #gradleVersion"() {
        given:
        buildFile << """
            task helloWorld {
                doLast {
                    logger.quiet 'Hello world!'
                }
            }
        """

        when:
        def result = GradleRunner.create()
            .withGradleVersion(gradleVersion)
            .withProjectDir(testProjectDir.root)
            .withArguments('helloWorld')
            .build()

        then:
        result.output.contains('Hello world!')
        result.task(":helloWorld").outcome == SUCCESS

        where:
        gradleVersion << ['2.6', '2.7']
    }
}
```

Feature support when testing with different Gradle versions

It is possible to use the GradleRunner to execute builds with Gradle 1.0 and later. However, some runner features are not supported on earlier versions. In such cases, the runner will throw an exception when attempting to use the feature.

The following table lists the features that are sensitive to the Gradle version being used.

Table 45.1. Gradle version compatibility

Feature	Minimum Version	Description
Inspecting executed tasks	2.5	Inspecting the executed tasks, using <code>BuildResult.getTasks()</code> and similar methods.
Plugin classpath injection	2.8	Injecting the code under test via <code>GradleRunner.withPluginClasspath(java.util.List.classpathElements)</code> .
Inspecting build output in debug mode	2.9	Inspecting the build's text output when run in debug mode via <code>BuildResult.getOutput()</code> .
Automatic plugin classpath injection	2.13	Injecting the code under test automatically via <code>GradleRunner.withPluginClasspath()</code> by using the <code>GradlePluginDevelopment</code> plugin.

45.2.4. Debugging build logic

The runner uses the Tooling API to execute builds. An implication of this is that the builds are executed in a separate process (i.e. not the same process executing the tests). Therefore, executing your *tests* in debug mode does not allow you to debug your build logic as you may expect. Any breakpoints set in your IDE will not be tripped by the code being exercised by the test build.

The TestKit provides two different ways to enable the debug mode:

- Setting “`org.gradle.testkit.debug`” system property to `true` for the JVM *using* the `GradleRunner` (i.e. not the build being executed with the runner);
- Calling the `GradleRunner.withDebug(boolean)` method.

The system property approach can be used when it is desirable to enable debugging support without making an adhoc change to the runner configuration. Most IDEs offer the capability to set JVM system properties for test execution, and such a feature can be used to set this system property.

Part V. Building JVM projects

46.1. The Java plugin

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 add code to your build script to do so.

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 some 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 unit 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.

46.2. A basic Java project

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

Example 46.1. Using the Java plugin

build.gradle

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

Note: The code for this example can be found at `samples/java/quickstart` in the ‘-all’ distribution of Gradle.

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 tasks** to list the tasks of a project. This will let you see the tasks that the Java plugin has added to your project.

46.2.1. 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 46.2. 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 checks plugin, this task will also run Checkstyle against your source code.

46.2.2. 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 46.3. 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 46.4. 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 7, *Dependency Management Basics*.

46.2.3. Customizing 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 46.5. Customization of MANIFEST.MF

build.gradle

```
sourceCompatibility = 1.7
version = '1.0'
jar {
    manifest {
        attributes '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 customize 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:

Example 46.6. Adding a test system property

build.gradle

```
test {
    systemProperties 'property': 'value'
}
```

What properties are available?

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

46.2.4. 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 46.7. Publishing the JAR file

build.gradle

```
uploadArchives {
    repositories {
        flatDir {
            dirs 'repos'
        }
    }
}
```

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

46.2.5. Creating an Eclipse project

To create the Eclipse-specific descriptor files, like `.project`, you need to add another plugin to your build file:

Example 46.8. Eclipse plugin

build.gradle

```
apply plugin: 'eclipse'
```

Now execute **gradle eclipse** command to generate Eclipse project files. More information about the `eclipse` task can be found in Chapter 66, *The Eclipse Plugins*.

46.2.6. Summary

Here's the complete build file for our sample:

Example 46.9. Java example - complete build file

build.gradle

```
apply plugin: 'java'
apply plugin: 'eclipse'

sourceCompatibility = 1.7
version = '1.0'
jar {
    manifest {
        attributes '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 {
    systemProperties 'property': 'value'
}

uploadArchives {
    repositories {
        flatDir {
            dirs 'repos'
        }
    }
}
```

46.3. Multi-project Java build

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

Example 46.10. Multi-project build - hierarchical layout

Build layout

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

Note: The code for this example can be found at `samples/java/multiproject` in the ‘-all’ distribution of Gradle.

Here we have four 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`. Project `services/shared` has code that depends on the `shared` project.

46.3.1. 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 46.11. Multi-project build - settings.gradle file

settings.gradle

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

You can find out more about the settings file in Chapter 26, *Multi-project Builds*.

46.3.2. 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 46.12. Multi-project build - common configuration

build.gradle

```
subprojects {
    apply plugin: 'java'
    apply plugin: 'eclipse-wtp'

    repositories {
        mavenCentral()
    }

    dependencies {
        testCompile 'junit:junit:4.12'
    }

    version = '1.0'

    jar {
        manifest.attributes provider: 'gradle'
    }
}
```

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.

Also note that these plugins are only applied within the `subprojects` section, not at the root level, so the root build will not expect to find Java source files in the root project, only in the subprojects.

46.3.3. 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 `shared` project. Due to this dependency, Gradle will ensure that project `shared` always gets built before project `api`.

Example 46.13. Multi-project build - dependencies between projects

api/build.gradle

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

See Section 26.7.1, “Disabling the build of dependency projects” for how to disable this functionality.

46.3.4. Creating a distribution

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

Example 46.14. Multi-project build - distribution file

api/build.gradle

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

artifacts {
    archives dist
}
```

46.4. Where to next?

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. You can find out more about the Java plugin in Chapter 47, *The Java Plugin*, and you can find more sample Java projects in the `samples/java` directory in the Gradle distribution.

Otherwise, continue on to Chapter 7, *Dependency Management Basics*.

47

The Java Plugin

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

47.1. Usage

To use the Java plugin, include the following in your build script:

Example 47.1. Using the Java plugin

build.gradle

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

47.2. Source sets

The Java plugin introduces the concept of a *source set*. A source set is simply 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.

One use for source sets is to group source files into logical groups which describe their purpose. For example, you might use a source set to define an integration test suite, or you might use separate source sets to define the API and implementation classes of your project.

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 test source code, which is compiled and executed using JUnit or TestNG. These can be unit tests, integration tests, acceptance tests, or any combination that is useful to you.

47.3. Tasks

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

Table 47.1. Java plugin - tasks

Task name	Depends on	Type	Description
-----------	------------	------	-------------

compileJava	All tasks which produce the compile classpath. This includes the jar task for project dependencies included in the compile configuration.	JavaCompile	Compiles production Java source files using javac.
processResources	-	Copy	Copies production resources into the production resources directory.
classes	The compileJava task and the processResources task. Some plugins add additional compilation tasks.	Task	Assembles the production classes and resources directories.
compileTestJava	compile, plus all tasks which produce the test compile classpath.	JavaCompile	Compiles test Java source files using javac.
processTestResources	-	Copy	Copies test resources into the test resources directory.
testClasses	compileTestJava task and processTestResources task. Some plugins add additional test compilation tasks.	Task	Assembles the test classes and resources directories.
jar	compile	Jar	Assembles the JAR file
javadoc	compile	Javadoc	Generates API documentation for the production Java source, using Javadoc
test	compile, compileTest, plus all tasks which produce the test runtime classpath.	Test	Runs the unit tests using JUnit or TestNG.

<code>uploadArchives</code>	The tasks which produce the artifacts in the archives configuration, including <code>jar</code> .	Upload	Uploads artifacts in the archives configuration, including the JAR file.
<code>clean</code>	-	Delete	Deletes the project build directory.
<code>cleanTaskName</code>	-	Delete	Deletes files created by specified task. <code>cleanJar</code> will delete the JAR file created by the <code>jar</code> task, and <code>cleanTest</code> will delete the test results created by the <code>test</code> task.

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

Table 47.2. Java plugin - source set tasks

Task name	Depends on	Type	Description
<code>compileSourceSetJava</code>	Tasks which produce the source set's compile classpath.	JavaCompile	Compiles the given source set's Java source files using javac.
<code>processSourceSetResources</code>		Copy	Copies the given source set's resources into the resources directory.
<code>sourceSetClasses</code>	Executes <code>compileSourceSetJava</code> task and the <code>processSourceSetResources</code> task. Some plugins add additional compilation tasks for the source set.	Class	Assembles the given source set's classes and resources directories.

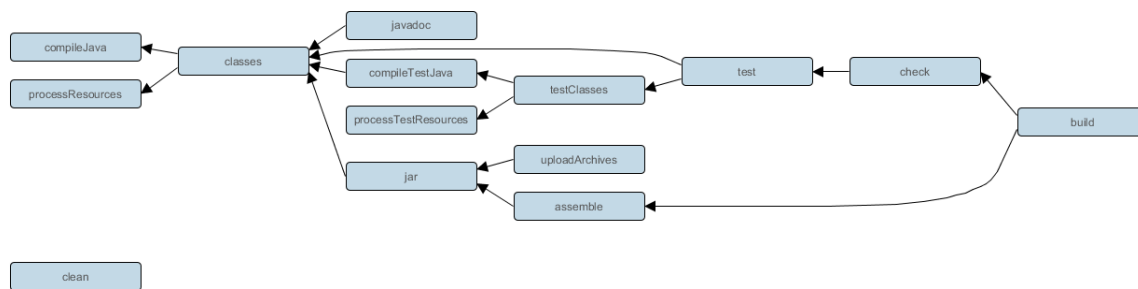
The Java plugin also adds a number of tasks which form a lifecycle for the project:

Table 47.3. 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	check and assemble	Task	Performs a full build of the project.
buildNeeded	build and buildNeeded 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	build and buildDependents 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.
buildConfigName	The tasks which produce the artifacts in configuration <i>ConfigName</i> .	Task	Assembles the artifacts in the specified configuration. The task is added by the Base plugin which is implicitly applied by the Java plugin.
uploadConfigName	The tasks which uploads the artifacts in configuration <i>ConfigName</i> .	Upload	Assembles and uploads the artifacts in the specified configuration. The task is added by the Base plugin which is implicitly applied by the Java plugin.

The following diagram shows the relationships between these tasks.

Figure 47.1. Java plugin - tasks



47.4. 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 47.4. 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

47.4.1. 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 47.2. Custom Java source layout

build.gradle

```

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

```

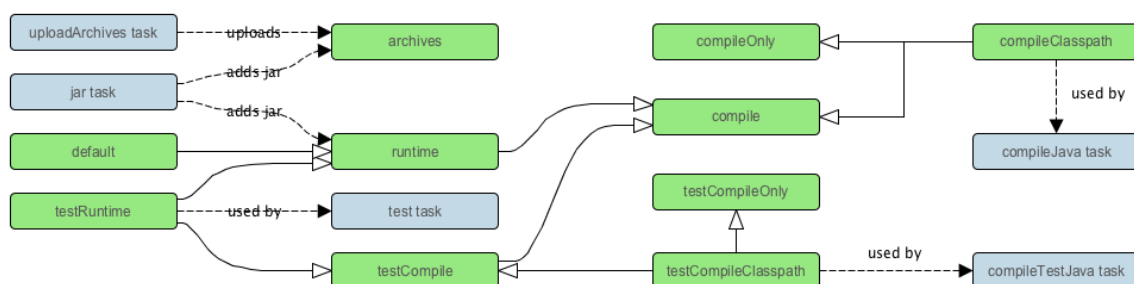
47.5. Dependency management

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`.

Table 47.5. Java plugin - dependency configurations

Name	Extends	Used by tasks	Meaning
<code>compile</code>	-	-	Compile time dependencies
<code>compileOnly</code>	-	-	Compile time only dependencies, not used at runtime
<code>compileClasspath</code>	<code>compile</code> , <code>compileOnly</code>	<code>compileJava</code>	Compile classpath, used when compiling source
<code>runtime</code>	<code>compile</code>	-	Runtime dependencies
<code>testCompile</code>	<code>compile</code>	-	Additional dependencies for compiling tests
<code>testCompileOnly</code>	-	-	Additional dependencies only for compiling tests, not used at runtime
<code>testCompileClasspath</code>	<code>testCompile</code> , <code>testCompileOnly</code>	<code>compileTestJava</code>	Test compile classpath, used when compiling test sources
<code>testRuntime</code>	<code>runtime</code> , <code>testCompile</code>	<code>test</code>	Additional dependencies for running tests only
<code>archives</code>	-	<code>uploadArchives</code>	Artifacts (e.g. jars) produced by this project
<code>default</code>	<code>runtime</code>	-	The default configuration used by a project dependency on this project. Contains the artifacts and dependencies required by this project at runtime.

Figure 47.2. Java plugin - dependency configurations



For each source set you add to the project, the Java plugin adds the following dependency configurations:

Table 47.6. Java plugin - source set dependency configurations

Name	Extends	Used by tasks	Meaning
<code>sourceSetCompile</code>	-	-	Compile time dependencies for the given source set
<code>sourceSetCompileOnly</code>		-	Compile time only dependencies for the given source set, not used at runtime
<code>sourceSetCompileClasspath</code>	<code>sourceSetCompile</code> , <code>sourceSetCompileOnly</code>	<code>compileSourceSetJava</code>	Compile classpath, used when compiling source
<code>sourceSetRuntime</code>	<code>sourceSetCompile</code>	-	Runtime dependencies for the given source set

47.6. 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.

Table 47.7. Java plugin - directory properties

Property name	Type	Default value	Description
<code>reportsDirName</code>	String	<code>reports</code>	The name of the directory to generate reports into, relative to the build directory.
<code>reportsDir</code>	File (read-only)	<code>buildDir/reportsDirName</code>	The directory to generate reports into.
<code>testResultsDirName</code>	String	<code>test-results</code>	The name of the directory to generate test result .xml files into, relative to the build directory.
<code>testResultsDir</code>	File (read-only)	<code>buildDir/testResultsDirName</code>	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	distributions	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 47.8. Java plugin - other properties

Property name	Type	Default value	Description
sourceSets	SourceSetContainer (read-only)	Not null	Contains the project's source sets.
sourceCompatibility	JavaVersion. Can also set using a String or a Number, e.g. '1.5' or 1.5.	version of the current JVM in use	Java version compatibility to use when compiling Java source.
targetCompatibility	JavaVersion. Can also set using a String or Number, e.g. '1.5' or 1.5.	<i>sourceCompatibility</i>	Java version to generate classes for.
archivesBaseName	String	<i>projectName</i>	The basename to use for archives, such as JAR or ZIP files.
manifest	Manifest	an empty manifest	The manifest to include in all JAR files.

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

47.7. 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 { }` script block, which you can pass a closure to configure the source set container. The source set container works pretty much the same way as other containers, such as `tasks`.

Example 47.3. Accessing a source set

build.gradle

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

// Iterate over the source sets
sourceSets.all {
    println 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 47.4. Configuring the source directories of a source set

build.gradle

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

47.7.1. 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 47.9. 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.
output	SourceSetOutput (read-only)	Not null	The output files of the source set, containing its compiled classes and resources.
output.classesDir	File	<i>buildDir/classes/name</i>	The directory to generate the classes of this source set into.
output.resourcesDir	File	<i>buildDir/resources/name</i>	The directory to generate the resources of this source set into.
compileClasspath	FileCollection	<i>compileSourceSet</i> configuration.	The classpath to use when compiling the source files of this source set.
runtimeClasspath	FileCollection	<i>output + runtimeSourceSet</i> configuration.	The classpath to use when executing the classes of this source set.

java	SourceDirectorySet (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 Section 20.5, “Specifying a set of input files”.	[projectDir/src/main/java]	The source directories containing the Java source files of this source set.
resources	SourceDirectorySet (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 20.5, “Specifying a set of input files”.	<code>[projectDir/src/main/resources]</code>	The resource directories containing the resources of this source set.
<code>allJava</code>	<code>SourceDirectorySet</code> (read-only)	<code>java</code>	All . java files of this source set. Some plugins, such as the Groovy plugin, add additional Java source files to this collection.
<code>allSource</code>	<code>SourceDirectorySet</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.

47.7.2. Defining new source sets

To define a new source set, you simply reference it in the `sourceSets { }` block. Here's an example:

Example 47.5. Defining a source set

build.gradle

```
sourceSets {  
    intTest  
}
```

When you define a new source set, the Java plugin adds some dependency configurations for the source set, as shown in Table 47.6, “Java plugin - source set dependency configurations”. You can use these configurations to define the compile and runtime dependencies of the source set.

Example 47.6. Defining source set dependencies

build.gradle

```
sourceSets {  
    intTest  
}  
  
dependencies {  
    intTestCompile 'junit:junit:4.12'  
    intTestRuntime 'org.ow2.asm:asm-all:4.0'  
}
```

The Java plugin also adds a number of tasks which assemble the classes for the source set, as shown in Table 47.2, “Java plugin - source set tasks”. For example, for a source set called `intTest`, compiling the classes for this source set is done by running **gradle intTestClasses**.

Example 47.7. Compiling a source set

Output of **gradle intTestClasses**

```
> gradle intTestClasses  
:compileIntTestJava  
:processIntTestResources  
:intTestClasses
```

```
BUILD SUCCESSFUL
```

```
Total time: 1 secs
```

47.7.3. Some source set examples

Adding a JAR containing the classes of a source set:

Example 47.8. Assembling a JAR for a source set

build.gradle

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

Generating Javadoc for a source set:

Example 47.9. 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 47.10. Running tests in a source set

build.gradle

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

47.8. 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 47.10. Java plugin - Javadoc properties

Task Property	Type	Default Value
<code>classpath</code>	<code>FileCollection</code>	<code>sourceSets.main.output + sourceSets.ma:</code>
<code>source</code>	<code>FileTree</code> . Can set using anything described in Section 20.5, “Specifying a set of input files”.	<code>sourceSets.main.allJava</code>
<code>destinationDir</code>	<code>File</code>	<code>docsDir/javadoc</code>
<code>title</code>	<code>String</code>	The name and version of the project

47.9. Clean

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

Table 47.11. Java plugin - Clean properties

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

47.10. 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 20.6, “Copying files”.

Table 47.12. Java plugin - ProcessResources properties

Task Property	Type	Default Value
<code>srcDirs</code>	Object. Can set using anything described in Section 20.5, “Specifying a set of input files”.	<code>sourceSet.resources</code>
<code>destinationDir</code>	File. Can set using anything described in Section 20.1, “Locating files”.	<code>sourceSet.output.resources</code>

47.11. CompileJava

The Java plugin adds a `JavaCompile` instance for each source set in the project. Some of the most common configuration options are shown below.

Table 47.13. Java plugin - Compile properties

Task Property	Type	Default Value
<code>classpath</code>	<code>FileCollection</code>	<code>sourceSet.compileClasspath</code>
<code>source</code>	<code>FileTree</code> . Can set using anything described in Section 20.5, “Specifying a set of input files”.	<code>sourceSet.java</code>
<code>destinationDir</code>	File.	<code>sourceSet.output.classesDir</code>

By default, the Java compiler runs in the Gradle process. Setting `options.fork` to `true` causes compilation to occur in a separate process. In the case of the Ant `javac` task, this means that a new process will be forked for each compile task, which can slow down compilation. Conversely, Gradle's direct compiler integration (see above) will reuse the same compiler process as much as possible. In both cases, all fork options specified with `options.forkOptions` will be honored.

47.12. Incremental Java compilation

Starting with Gradle 2.1, it is possible to compile Java incrementally. See the `JavaCompile` task for information on how to enable it.

Main goals for incremental compilations are:

- Avoid wasting time compiling source classes that don't have to be compiled. This means faster builds, especially when a change to a source class or a jar does not incur recompilation of many source classes that depend on the changed input.

- Change as few output classes as possible. Classes that don't need to be recompiled remain unchanged in the output directory. An example scenario when this is really useful is using JRebel - the fewer output classes are changed the quicker the jvm can use refreshed classes.

The incremental compilation at a high level:

- The detection of the correct set of stale classes is reliable at some expense of speed. The algorithm uses bytecode analysis and deals gracefully with compiler optimizations (inlining of non-private constants), transitive class dependencies, etc. Example: When a class with a public constant changes, we eagerly compile classes that use the same constants to avoid problems with constants inlined by the compiler.
- To make incremental compilation fast, we cache class analysis results and jar snapshots. The initial incremental compilation can be slower due to the cold caches.

Known issues

- If a compile task fails due to a compile error, it will do a full compilation again the next time it is invoked.
- Because of type erasure, the incremental compiler is not able to recognize when a type is only used in a type parameter, and never actually used in the code. For example, imagine that you have the following code: `List<? extends A> list = Lists.newArrayList();` but that no member of A is in practice used in the code, then changes to A will not trigger recompilation of the class. In practice, this should very rarely be an issue.

47.13. Compile avoidance

If a dependent project has changed in an ABI-compatible way (only its private API has changed), then Java compilation tasks will be up-to-date. This means that if project A depends on project B and a class in B is changed in an ABI-compatible way (typically, changing only the body of a method), then Gradle won't recompile A.

Some of the types of changes that do not affect the public API and are ignored:

- Changing a method body
- Changing a comment
- Adding, removing or changing private methods, fields, or inner classes
- Adding, removing or changing a resource
- Changing the name of jars or directories in the classpath
- Renaming a parameter

Compile-avoidance is deactivated if annotation processors are found on the compile classpath, because for annotation processors the implementation details matter. To better separate these concerns, it's recommended to declare annotation processors separately: the `CompileOptions` for the `JavaCompile` task type define a `annotationProcessorPath` property that can be used to declare annotation processors. It's recommended to use a distinct configuration for annotation processors:

Example 47.11. Declaring annotation processors

build.gradle

```
configurations {
    apt
}
dependencies {
    // The dagger compiler and its transitive dependencies will only be found on ann
    apt 'com.google.dagger:dagger-compiler:2.8'

    // And we still need the Dagger annotations on the compile classpath itself
    implementation 'com.google.dagger:dagger:2.8'
}

compileJava {
    options.annotationProcessorPath = configurations.apt
}
```

47.14. Test

The `test` task is an instance of `Test`. It automatically detects and executes all unit tests in the `test` source set. It also generates a report once test execution is complete. JUnit and TestNG are both supported. Have a look at `Test` for the complete API.

47.14.1. Test execution

Tests are executed in a separate JVM, isolated from the main build process. The `Test` task's API allows you some control over how this happens.

There are a number of properties which control how the test process is launched. This includes things such as system properties, JVM arguments, and the Java executable to use.

You can specify whether or not to execute your tests in parallel. Gradle provides parallel test execution by running multiple test processes concurrently. Each test process executes only a single test at a time, so you generally don't need to do anything special to your tests to take advantage of this. The `maxParallelForks` property specifies the maximum number of test processes to run at any given time. The default is 1, that is, do not execute the tests in parallel.

The test process sets the `org.gradle.test.worker` system property to a unique identifier for that test process, which you can use, for example, in files names or other resource identifiers.

You can specify that test processes should be restarted after it has executed a certain number of test classes. This can be a useful alternative to giving your test process a very large heap. The `forkEvery` property specifies the maximum number of test classes to execute in a test process. The default is to execute an unlimited number of tests in each test process.

The task has an `ignoreFailures` property to control the behavior when tests fail. The `Test` task always executes every test that it detects. It stops the build afterwards if `ignoreFailures` is false and there are failing tests. The default value of `ignoreFailures` is false.

The `testLogging` property allows you to configure which test events are going to be logged and at which detail level. By default, a concise message will be logged for every failed test. See `TestLoggingContainer` for how to tune test logging to your preferences.

47.14.2. Debugging

The test task provides a `Test.getDebug()` property that can be set to launch to make the JVM wait for a debugger to attach to port 5005 before proceeding with test execution.

This can also be enabled at invocation time via the `--debug-jvm` task option (since Gradle 1.12).

47.14.3. Test filtering

Starting with Gradle 1.10, it is possible to include only specific tests, based on the test name pattern. Filtering is a different mechanism than test class inclusion / exclusion that will be described in the next few paragraphs (`-Dtest.single`, `test.include` and friends). The latter is based on files, e.g. the physical location of the test implementation class. File-level test selection does not support many interesting scenarios that are possible with test-level filtering. Some of them Gradle handles now and some will be satisfied in future releases:

- Filtering at the level of specific test methods; executing a single test method
- Filtering based on custom annotations (future)
- Filtering based on test hierarchy; executing all tests that extend a certain base class (future)
- Filtering based on some custom runtime rule, e.g. particular value of a system property or some static state (future)

Test filtering feature has following characteristic:

- Fully qualified class name or fully qualified method name is supported, e.g. `“org.gradle.SomeTest”`, `“org.gradle.SomeTest.someMethod”`
- Wildcard `*` is supported for matching any characters
- Command line option `“--tests”` is provided to conveniently set the test filter. Especially useful for the classic 'single test method execution' use case. When the command line option is used, the inclusion filters declared in the build script are ignored. It is possible to supply multiple `“--tests”` options and tests matching any of those patterns will be included.
- Gradle tries to filter the tests given the limitations of the test framework API. Some advanced, synthetic tests may not be fully compatible with filtering. However, the vast majority of tests and use cases should be handled neatly.
- Test filtering supersedes the file-based test selection. The latter may be completely replaced in future. We will grow the the test filtering api and add more kinds of filters.

Example 47.12. Filtering tests in the build script

build.gradle

```
test {
    filter {
        //include specific method in any of the tests
        includeTestsMatching "*UiCheck"

        //include all tests from package
        includeTestsMatching "org.gradle.internal.*"

        //include all integration tests
        includeTestsMatching "*IntegTest"
    }
}
```

For more details and examples please see the `TestFilter` reference.

Some examples of using the command line option:

- `gradle test --tests org.gradle.SomeTest.someSpecificFeature`
- `gradle test --tests *SomeTest.someSpecificFeature`
- `gradle test --tests *SomeSpecificTest`
- `gradle test --tests *SomeSpecificTestSuite`
- `gradle test --tests all.in.specific.package*`
- `gradle test --tests *IntegTest`
- `gradle test --tests *IntegTest*ui*`
- `gradle test --tests "com.example.MyTestSuite"`
- `gradle test --tests "com.example.ParameterizedTest"`
- `gradle test --tests "*ParameterizedTest.foo"`
- `gradle test --tests "*ParameterizedTest.*[2]"`
- `gradle someTestTask --tests *UiTest someOtherTestTask --tests *WebTest*ui`

47.14.4. Single test execution via System Properties

This mechanism has been superseded by 'Test Filtering', described above.

Setting a system property of `taskName.single = testNamePattern` will only execute tests that match the specified `testNamePattern`. The `taskName` can be a full multi-project path like `":sub1:sub2:test"` or just the task name. The `testNamePattern` will be used to form an include pattern of `"**/testNamePattern*.class"`. If no tests with this pattern can be found an exception is thrown. This is to shield you from false security. If tests of more than one subproject are executed, the pattern is applied to each subproject. An exception is thrown if no tests can be found for a particular subproject. In such a case you can use the path notation of the pattern, so that the pattern is applied only to the test task of a specific subproject. Alternatively you can specify the fully qualified task name to be executed. You can also specify multiple patterns. Examples:

- `gradle -Dtest.single=ThisUniquelyNamedTest test`

- `gradle -Dtest.single=a/b/ test`
- `gradle -DintegTest.single=*IntegrationTest integTest`
- `gradle -Dtest.single=:proj1:test:Customer build`
- `gradle -DintegTest.single=c/d/ :proj1:integTest`

47.14.5. Test detection

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:

- Class or a super class extends `TestCase` or `GroovyTestCase`
- Class or a super class is annotated with `@RunWith`
- Class or a super class contain a method annotated with `@Test`

When using TestNG, we scan for methods annotated with `@Test`.

Note that abstract classes are not executed. Gradle also scans up the inheritance tree into jar files on the test classpath.

If you don't want to use test class detection, you can disable it by setting `scanForTestClasses` to `false`. This will make the test task only use includes / excludes to find test classes. If `scanForTestClasses` is `false` and no include / exclude patterns are specified, the defaults are `**/*Tests.class`, `**/*Test.class` and `**/Abstract*.class` for include and exclude, respectively.

47.14.6. Test grouping

JUnit and TestNG allows sophisticated groupings of test methods.

For grouping JUnit test classes and methods JUnit 4.8 introduces the concept of categories. ^[23] The `test` task allows the specification of the JUnit categories you want to include and exclude.

Example 47.13. JUnit Categories

build.gradle

```
test {
    useJUnit {
        includeCategories 'org.gradle.junit.CategoryA'
        excludeCategories 'org.gradle.junit.CategoryB'
    }
}
```

The TestNG framework has a quite similar concept. In TestNG you can specify different test groups. ^[24] The test groups that should be included or excluded from the test execution can be configured in the test task.

Example 47.14. Grouping TestNG tests

build.gradle

```
test {
    useTestNG {
        excludeGroups 'integrationTests'
        includeGroups 'unitTests'
    }
}
```

47.14.7. Test execution order in TestNG

TestNG allows explicit control of the execution order of tests.

The `preserveOrder` property controls whether tests are executed in deterministic order. Preserving the order guarantees that the complete test (including `@BeforeXXX` and `@AfterXXX`) is run in a test thread before the next test is run. While preserving the order of tests is the default behavior when directly working with `testng.xml` files, the TestNG API, that is used for running tests programmatically, as well as Gradle's TestNG integration execute tests in unpredictable order by default. ^[25] Preserving the order of tests was introduced with TestNG version 5.14.5. Setting the `preserveOrder` property to `true` for an older TestNG version will cause the build to fail.

Example 47.15. Preserving order of TestNG tests

build.gradle

```
test {
    useTestNG {
        preserveOrder true
    }
}
```

The `groupByInstance` property controls whether tests should be grouped by instances. Grouping by instances will result in resolving test method dependencies for each instance instead of running the dependees of all instances before running the dependants. The default behavior is not to group tests by instances. ^[26] Grouping tests by instances was introduced with TestNG version 6.1. Setting the `groupByInstance` property to `true` for an older TestNG version will cause the build to fail.

Example 47.16. Grouping TestNG tests by instances

build.gradle

```
test {
    useTestNG {
        groupByInstances true
    }
}
```

47.14.8. Test reporting

The `Test` task generates the following results by default.

- An HTML test report.
- The results in an XML format that is compatible with the Ant JUnit report task. This format is supported by many other tools, such as CI servers.
- Results in an efficient binary format. The task generates the other results from these binary results.

There is also a stand-alone `TestReport` task type which can generate the HTML test report from the binary results generated by one or more `Test` task instances. To use this task type, you need to define a `destinationDir` and the test results to include in the report. Here is a sample which generates a combined report for the unit tests from subprojects:

Example 47.17. Creating a unit test report for subprojects

build.gradle

```
subprojects {
    apply plugin: 'java'

    // Disable the test report for the individual test task
    test {
        reports.html.enabled = false
    }
}

task testReport(type: TestReport) {
    destinationDir = file("$buildDir/reports/allTests")
    // Include the results from the `test` task in all subprojects
    reportOn subprojects*.test
}
```

You should note that the `TestReport` type combines the results from multiple test tasks and needs to aggregate the results of individual test classes. This means that if a given test class is executed by multiple test tasks, then the test report will include executions of that class, but it can be hard to distinguish individual executions of that class and their output.

TestNG parameterized methods and reporting

TestNG supports parameterizing test methods, allowing a particular test method to be executed multiple times with different inputs. Gradle includes the parameter values in its reporting of the test method execution.

Given a parameterized test method named `aTestMethod` that takes two parameters, it will be reported with the name: `aTestMethod(toStringValueOfParam1, toStringValueOfParam2)`. This makes identifying the parameter values for a particular iteration easy.

47.14.9. Convention values

Table 47.14. Java plugin - test properties

Task Property	Type	Default Value
testClassesDir	File	sourceSets.test.output.classesDir
classpath	FileCollection	sourceSets.test.runtimeClasspath
testResultsDir	File	testResultsDir
testReportDir	File	testReportDir

47.15. 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. You can learn more about how to work with archives in Section 20.8, “Creating archives” and artifact configurations in Chapter 32, *Publishing artifacts*.

47.15.1. Manifest

Each `jar` or `war` object has a `manifest` property with a separate instance of `Manifest`. When the archive is generated, a corresponding `MANIFEST.MF` file is written into the archive.

Example 47.18. Customization of MANIFEST.MF

build.gradle

```
jar {
    manifest {
        attributes("Implementation-Title": "Gradle",
                  "Implementation-Version": version)
    }
}
```

You can create stand alone instances of a `Manifest`. You can use that for example, to share manifest information between jars.

Example 47.19. Creating a manifest object.

build.gradle

```
ext.sharedManifest = manifest {
    attributes("Implementation-Title": "Gradle",
              "Implementation-Version": version)
}
task fooJar(type: Jar) {
    manifest = project.manifest {
        from sharedManifest
    }
}
```

You can merge other manifests into any Manifest object. The other manifests might be either described by a file path or, like in the example above, by a reference to another Manifest object.

Manifests are merged in the order they are declared by the `from` statement. If the base manifest and the merged manifest both define values for the same key, the merged manifest wins by default. You can fully customize the merge behavior by adding `eachEntry` actions in which you have access to a `ManifestMergeDetails` instance for each entry of the resulting manifest. The merge is not immediately triggered by the `from` statement. It is done lazily, either when generating the jar, or by calling `write` or `effectiveManifest`

You can easily write a manifest to disk.

Example 47.20. Separate MANIFEST.MF for a particular archive

build.gradle

```
task barJar(type: Jar) {
    manifest {
        attributes key1: 'value1'
        from sharedManifest, 'src/config/basemanifest.txt'
        from('src/config/javabasemanifest.txt',
            'src/config/libbasemanifest.txt') {
            eachEntry { details ->
                if (details.baseValue != details.mergeValue) {
                    details.value = baseValue
                }
                if (details.key == 'foo') {
                    details.exclude()
                }
            }
        }
    }
}
```

build.gradle

```
jar.manifest.writeTo("$buildDir/mymanifest.mf")
```

47.16. Uploading

How to upload your archives is described in Chapter 32, *Publishing artifacts*.

47.17. Compiling and testing for Java 6

Gradle can only run on Java version 7 or higher. Gradle 3.x still supports compiling, testing, generating Javadoc and executing applications for Java 6. Java 5 is not supported.

To use Java 6 the following tasks need to be configured:

- `JavaCompile` task to fork and use the correct `javac` executable
- `Javadoc` task to use the correct `javadoc` executable
- `Test` and the `JavaExec` task to use the correct `java` executable.

The following sample shows how the `build.gradle` needs to be adjusted. In order to be able to make the build machine-independent, the location of the Java 6 home should be configured in `GRADLE_USER_HOME/gradle` [27] in the users home directory on each developer machine, as shown in the example.

Example 47.21. Configure Java 6 build

gradle.properties

```
# in $HOME/.gradle/gradle.properties
java6Home=/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
```

build.gradle

```
sourceCompatibility = 1.6

assert hasProperty('java6Home') : "Set the property 'java6Home' in your your gradle."
def javaExecutablesPath = new File(java6Home, 'bin')
def javaExecutables = [:].withDefault { execName ->
    def executable = new File(javaExecutablesPath, execName)
    assert executable.exists() : "There is no ${execName} executable in ${javaExecutablesPath}"
    executable
}

tasks.withType(AbstractCompile) {
    options.with {
        fork = true
        forkOptions.executable = javaExecutables.javac
    }
}

tasks.withType(Javadoc) {
    executable = javaExecutables.javadoc
}

tasks.withType(Test) {
    executable = javaExecutables.java
}

tasks.withType(JavaExec) {
    executable = javaExecutables.java
}
```

[23] The JUnit wiki contains a detailed description on how to work with JUnit categories:
<https://github.com/junit-team/junit/wiki/Categories>.

[24] The TestNG documentation contains more details about test groups:
<http://testng.org/doc/documentation-main.html#test-groups>.

[25] The TestNG documentation contains more details about test ordering when working with `testng.xml` files: <http://testng.org/doc/documentation-main.html#testng-xml>.

[26] The TestNG documentation contains more details about grouping tests by instances:
<http://testng.org/doc/documentation-main.html#dependencies-with-annotations>.

[27] For more details on `gradle.properties` see Section 12.1, “Configuring the build environment via `gradle.properties`”

The Java Library Plugin

The Java Library plugin expands the capabilities of the Java plugin by providing specific knowledge about Java libraries. In particular, a Java library exposes an API to consumers (i.e., other projects using the Java or the Java Library plugin). All the source sets, tasks and configurations exposed by the Java plugin are implicitly available when using this plugin.

48.1. Usage

To use the Java Library plugin, include the following in your build script:

Example 48.1. Using the Java Library plugin

build.gradle

```
apply plugin: 'java-library'
```

48.2. API and implementation separation

The key difference between the standard Java plugin and the Java Library plugin is that the latter introduces the concept of an *API* exposed to consumers. A library is a Java component meant to be consumed by other components. It's a very common use case in multi-project builds, but also as soon as you have external dependencies.

The plugin exposes two configurations that can be used to declare dependencies: `api` and `implementation`. The `api` configuration should be used to declare dependencies which are exported by the library API, whereas the `implementation` configuration should be used to declare dependencies which are internal to the component.

Example 48.2. Declaring API and implementation dependencies

build.gradle

```
dependencies {  
    api 'commons-httpclient:commons-httpclient:3.1'  
    implementation 'org.apache.commons:commons-lang3:3.5'  
}
```

Dependencies appearing in the `api` configurations will be transitively exposed to consumers of the library, and as such will appear on the compile classpath of consumers. Dependencies found in the `implementation`

configuration will, on the other hand, not be exposed to consumers, and therefore not leak into the consumers' compile classpath. This comes with several benefits:

- dependencies do not leak into the compile classpath of consumers anymore, so you will never accidentally depend on a transitive dependency
- faster compilation thanks to reduced classpath size
- less recompilations when implementation dependencies change: consumers would not need to be recompiled
- cleaner publishing: when used in conjunction with the new `maven-publish` plugin, Java libraries produce POM files that distinguish exactly between what is required to compile against the library and what is required to use the library at runtime (in other words, don't mix what is needed to compile the library itself and what is needed to compile against the library).

The `compile` configuration still exists but should not be used as it will not offer the guarantees that the `api` and `implementation` configurations provide.

48.3. Recognizing API and implementation dependencies

This section will help you spot API and Implementation dependencies in your code using simple rules of thumb. Basically, an API dependency is a type that is exposed in the library binary interface, often referred to ABI (Application Binary Interface). This includes, but is not limited to:

- types used in super classes or interfaces
- types used in public method parameters, including generic parameter types (where *public* is something that is visible to compilers. I.e. , *public*, *protected* and *package private* members in the Java world)
- types used in public fields
- public annotation types

In opposition, any type that is used in the following list is irrelevant to the ABI, and therefore should be declared as `implementation` dependency:

- types exclusively used in method bodies
- types exclusively used in private members
- types exclusively found in internal classes (future versions of Gradle will let you declare which packages belong to the public API)

In the following sample, we can make the difference between an API dependency and an implementation dependency:

Example 48.3. Making the difference between API and implementation

src/main/java/org/gradle/HttpClientWrapper.java

```
// The following types can appear anywhere in the code
// but say nothing about API or implementation usage
import org.apache.commons.httpclient.*;
import org.apache.commons.httpclient.methods.*;
import org.apache.commons.lang3.exception.ExceptionUtils;
import java.io.IOException;
import java.io.UnsupportedEncodingException;

public class HttpClientWrapper {

    private final HttpClient client; // private member: implementation details

    // HttpClient is used as a parameter of a public method
    // so "leaks" into the public API of this component
    public HttpClientWrapper(HttpClient client) {
        this.client = client;
    }

    // public methods belongs to your API
    public byte[] doRawGet(String url) {
        GetMethod method = new GetMethod(url);
        try {
            int statusCode = doGet(method);
            return method.getResponseBody();
        } catch (Exception e) {
            ExceptionUtils.rethrow(e); // this dependency is internal only
        } finally {
            method.releaseConnection();
        }
        return null;
    }

    // GetMethod is used in a private method, so doesn't belong to the API
    private int doGet(GetMethod method) throws Exception {
        int statusCode = client.executeMethod(method);
        if (statusCode != HttpStatus.SC_OK) {
            System.err.println("Method failed: " + method.getStatusLine());
        }
        return statusCode;
    }
}
```

We can see that our class imports third party classes, but imports alone won't tell us if a dependency is an API or implementation dependency. For this, we need to look at the methods. The public constructor of `HttpClientWrapper` uses `HttpClient` as a parameter, so it is exposed to consumers and therefore belongs to the API.

On the other hand, the `ExceptionUtils` type, coming from the `commons-lang` library, is only used in a method body, so it's an implementation dependency.

Therefore, we can deduce that `commons-httpclient` is an API dependency, whereas `commons-lang` is an implementation dependency, which directly translates into the build file:

Example 48.4. Declaring API and implementation dependencies

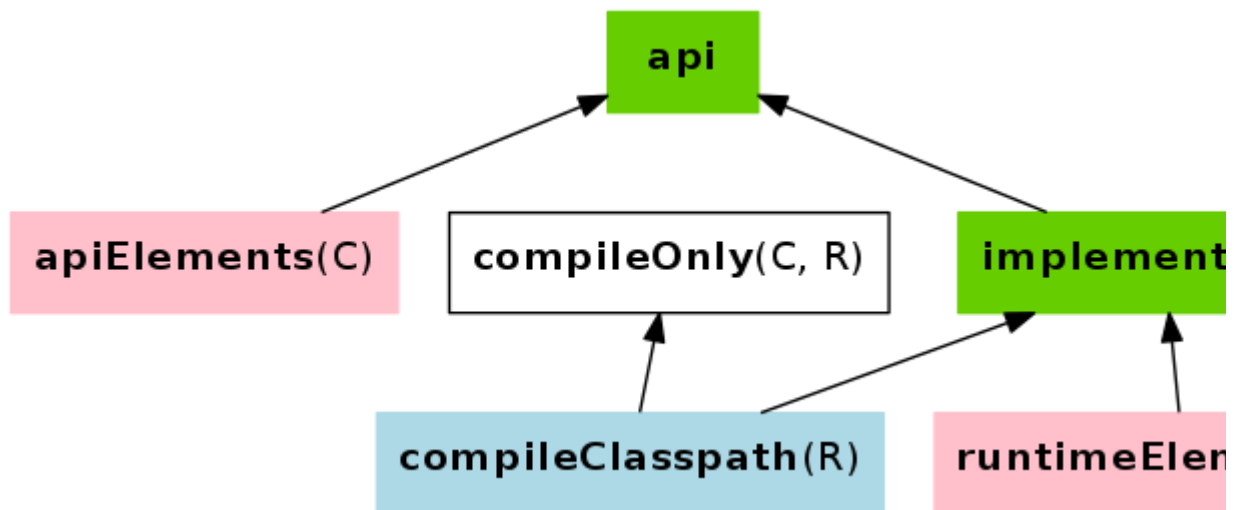
build.gradle

```
dependencies {  
    api 'commons-httpclient:commons-httpclient:3.1'  
    implementation 'org.apache.commons:commons-lang3:3.5'  
}
```

As a guideline, you should prefer the `implementation` configuration first: leakage of implementation types to consumers would then directly lead to a compile error of consumers, which would be solved either by removing the type from the public API, or promoting the dependency as an API dependency instead.

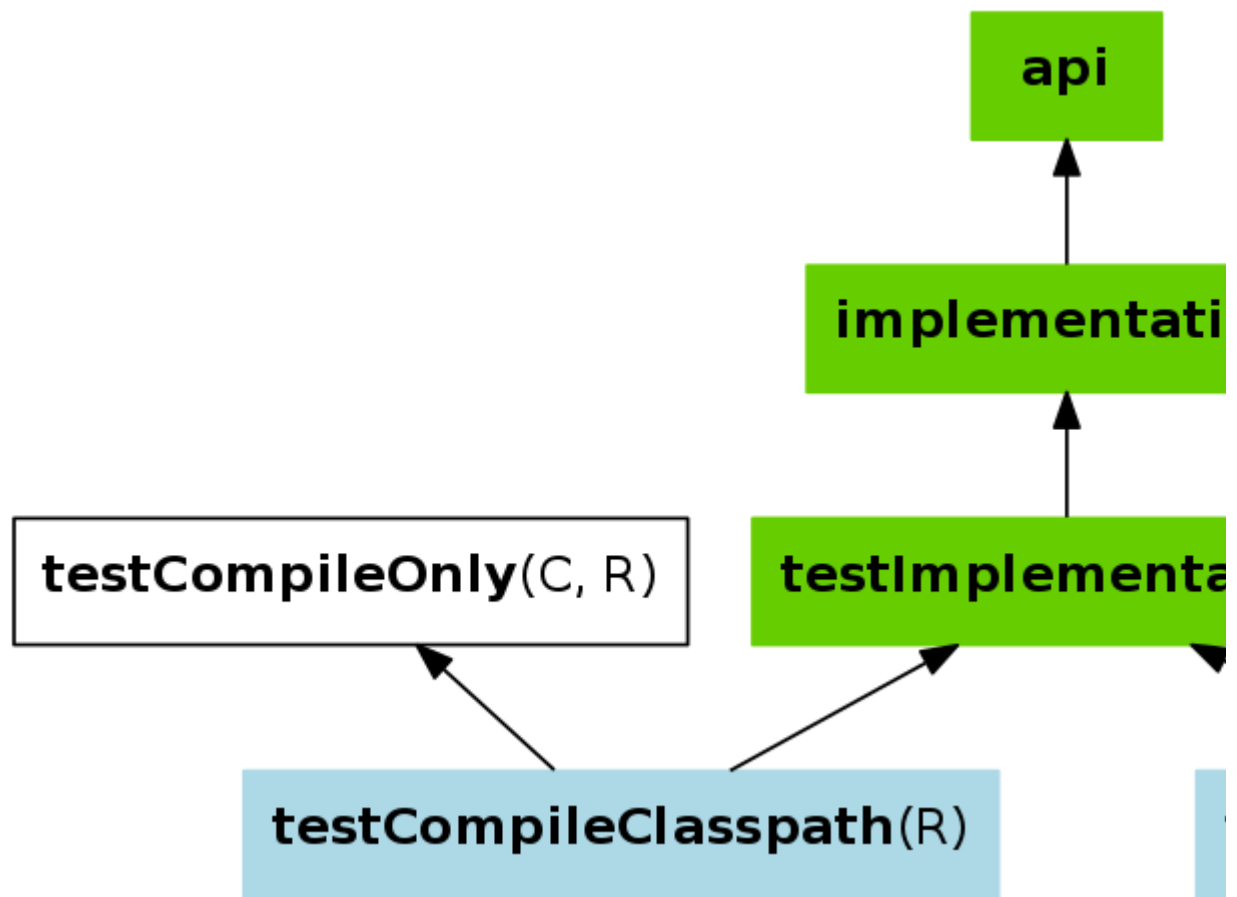
48.4. The Java Library plugin configurations

The following graph describes the main configurations setup when the Java Library plugin is in use.



- The configurations in *green* are the ones a user should use to declare dependencies
- The configurations in *pink* are the ones used when a component compiles, or runs against the library
- The configurations in *blue* are internal to the component, for its own use
- The configurations in *white* are configurations inherited from the Java plugin

And the next graph describes the test configurations setup:



The *compile*, *testCompile*, *runtime* and *testRuntime* configurations inherited from the Java plugin are still available but are deprecated. You should avoid using them, as they are only kept for backwards compatibility.

The role of each configuration is described in the following tables:

Table 48.1. Java Library plugin - configurations used to declare dependencies

Configuration name	Role	Can be consumed	Can be resolved	Description
api	Declaring API dependencies	no	no	This is where you should declare dependencies which are transitively exported to consumers, for compile.
implementation	Declaring implementation dependencies	no	no	This is where you should declare dependencies which are purely internal and not meant to be exposed to consumers.
compileOnly	Declaring compile only dependencies	yes	yes	This is where you should declare dependencies which are only required at compile time, but should not leak into the runtime. This typically includes dependencies which are shaded when found at runtime.
runtimeOnly	Declaring runtime dependencies	no	no	This is where you should declare dependencies which are only required at runtime, and not at compile time.
testImplementation	Test dependencies	no	no	This is where you should declare dependencies which are used to compile tests.
testCompileOnly	Declaring test compile only dependencies	yes	yes	This is where you should declare dependencies which are only required at test compile time, but should not leak into the runtime. This typically includes dependencies which are shaded when found at runtime.
testRuntimeOnly	Declaring test runtime dependencies	no	no	This is where you should declare dependencies which are only required at test runtime, and not at test compile time.

Table 48.2. Java Library plugin - configurations used by consumers

Configuration name	Role	Can be consumed	Can be resolved	Description
apiElements	For compiling against this library	yes	no	This configuration is meant to be used by consumers, to retrieve all the elements necessary to compile against this library. Unlike the default configuration, this doesn't leak implementation or runtime dependencies.
runtimeElements	For executing this library	yes	no	This configuration is meant to be used by consumers, to retrieve all the elements necessary to run against this library.

Table 48.3. Java Library plugin - configurations used by the library itself

Configuration name	Role	Can be consumed	Can be resolved	Description
compileClasspath	For compiling this library	no	yes	This configuration contains the compile classpath of this library, and is therefore used when invoking the java compiler to compile it.
runtimeClasspath	For executing this library	no	yes	This configuration contains the runtime classpath of this library
testCompileClasspath	For compiling the tests of this library	no	yes	This configuration contains the test compile classpath of this library.
testRuntimeClasspath	For executing tests of this library	no	yes	This configuration contains the test runtime classpath of this library

48.5. Known issues

48.5.1. Compatibility with other plugins

At the moment the Java Library plugin is only wired to behave correctly with the `java` plugin. Other plugins, such as the Groovy plugin, may not behave correctly. In particular, if the Groovy plugin is used in addition to the `java-library` plugin, then consumers may not get the Groovy classes when they consume the library. To workaround this, you need to explicitly wire the Groovy compile dependency, like this:

Example 48.5. Configuring the Groovy plugin to work with Java Library

a/build.gradle

```
configurations {
    apiElements {
        outgoing.variants.getByName('classes').artifact(
            file: compileGroovy.destinationDir,
            type: JavaPlugin.CLASS_DIRECTORY,
            builtBy: compileGroovy)
    }
}
```

48.5.2. Increased memory usage for consumers

When a project uses the Java Library plugin, consumers will use the output classes directory of this project directly on their compile classpath, instead of the jar file if the project uses the Java plugin. An indirect consequence is that up-to-date checking will require more memory, because Gradle will snapshot individual class files instead of a single jar. This may lead to increased memory consumption for large projects.

Web Application Quickstart

This chapter is a work in progress.

This chapter introduces the Gradle 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.

49.1. Building a WAR file

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

Example 49.1. War plugin

build.gradle

```
apply plugin: 'war'
```

Note: The code for this example can be found at `samples/webApplication/quickstart` in the ‘-all’ distribution of Gradle.

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, in the `WEB-INF/classes` and `WEB-INF/lib` directories, respectively.

49.2. Running your web application

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

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

Example 49.2. Running web application with Jetty plugin

build.gradle

```
apply plugin: 'jetty'
```

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 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.

49.3. Summary

You can find out more about the War plugin in Chapter 50, *The War Plugin* and the Jetty plugin in Chapter 52, *The Jetty Plugin*. You can find more sample Java projects in the `samples/webApplication` directory in the Gradle distribution.

50

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.

50.1. Usage

To use the War plugin, include the following in your build script:

Example 50.1. Using the War plugin

build.gradle

```
apply plugin: 'war'
```

50.2. Tasks

The War plugin adds the following tasks to the project.

Table 50.1. War plugin - tasks

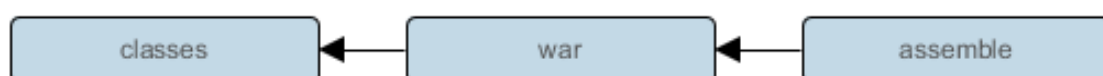
Task name	Depends on	Type	Description
war	compile	War	Assembles the application WAR file.

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

Table 50.2. War plugin - additional task dependencies

Task name	Depends on
assemble	war

Figure 50.1. War plugin - tasks



50.3. Project layout

Table 50.3. War plugin - project layout

Directory	Meaning
src/main/webapp	Web application sources

50.4. Dependency management

The War plugin adds two dependency configurations named `providedCompile` and `providedRuntime`. Those two 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`. Because this is a “provided” configuration, this means that neither of these dependencies will be added to your WAR, even if the `commons-codec` library is an explicit dependency of your `compile` configuration. If you don't want this transitive behavior, simply declare your provided dependencies like `commons-httpclient:`.

50.5. Convention properties

Table 50.4. 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.

50.6. 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 directory may of course contain a `WEB-INF` sub-directory, which may contain a `web.xml` file. Your compiled classes are compiled to `WEB-INF/classes`. All the dependencies of the runtime [28] configuration are copied to `WEB-INF/lib`.

The War class in the API documentation has additional useful information.

50.7. Customizing

Here is an example with the most important customization options:

Example 50.2. Customization of war plugin

build.gradle

```
configurations {
    moreLibs
}

repositories {
    flatDir { dirs "lib" }
    mavenCentral()
}

dependencies {
    compile module(":compile:1.0") {
        dependency ":compile-transitive-1.0@jar"
        dependency ":providedCompile-transitive:1.0@jar"
    }
    providedCompile "javax.servlet:servlet-api:2.5"
    providedCompile module(":providedCompile:1.0") {
        dependency ":providedCompile-transitive:1.0@jar"
    }
    runtime ":runtime:1.0"
    providedRuntime ":providedRuntime:1.0@jar"
    testCompile "junit:junit:4.12"
    moreLibs ":otherLib:1.0"
}

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

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

[28] The runtime configuration extends the compile configuration.

51

The Ear Plugin

The Ear plugin adds support for assembling web application EAR files. It adds a default EAR archive task. It doesn't require the Java plugin, but for projects that also use the Java plugin it disables the default JAR archive generation.

51.1. Usage

To use the Ear plugin, include the following in your build script:

Example 51.1. Using the Ear plugin

build.gradle

```
apply plugin: 'ear'
```

51.2. Tasks

The Ear plugin adds the following tasks to the project.

Table 51.1. Ear plugin - tasks

Task name	Depends on	Type	Description
ear	compile (only if the Java plugin is also applied)	Ear	Assembles the application EAR file.

The Ear plugin adds the following dependencies to tasks added by the base plugin.

Table 51.2. Ear plugin - additional task dependencies

Task name	Depends on
assemble	ear

51.3. Project layout

Table 51.3. Ear plugin - project layout

Directory	Meaning
src/main/application	Ear resources, such as a META-INF directory

51.4. Dependency management

The Ear plugin adds two dependency configurations: `deploy` and `earlib`. All dependencies in the `deploy` configuration are placed in the root of the EAR archive, and are *not* transitive. All dependencies in the `earlib` configuration are placed in the 'lib' directory in the EAR archive and *are* transitive.

51.5. Convention properties

Table 51.4. Ear plugin - directory properties

Property name	Type	Default value	Description
<code>appDirName</code>	String	<code>src/main/application</code>	The name of the application directory, in the generated EAR archive.
<code>libDirName</code>	String	<code>lib</code>	The name of the library directory, in the generated EAR archive.
<code>deploymentDescriptor</code>	<code>org.gradle.plugins.ear.descriptor.DeploymentDescriptor</code>	A deployment descriptor with sensible defaults named <code>application.xml</code> .	Metadata that describes the application. If this file is present in the application directory, then the existing descriptor will be used and the default descriptor will be ignored.

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

51.6. Ear

The default behavior of the Ear task is to copy the content of `src/main/application` to the root of the archive. If your application directory doesn't contain a `META-INF/application.xml` deployment descriptor then one will be generated for you.

The `Ear` class in the API documentation has additional useful information.

51.7. Customizing

Here is an example with the most important customization options:

Example 51.2. Customization of ear plugin

build.gradle

```
apply plugin: 'ear'
apply plugin: 'java'

repositories { mavenCentral() }

dependencies {
    // The following dependencies will be the ear modules and
    // will be placed in the ear root
    deploy project(path: ':war', configuration: 'archives')

    // The following dependencies will become ear libs and will
    // be placed in a dir configured via the libDirName property
    earlib group: 'log4j', name: 'log4j', version: '1.2.15', ext: 'jar'
}

ear {
    appDirName 'src/main/app' // use application metadata found in this folder
    // put dependent libraries into APP-INF/lib inside the generated EAR
    libDirName 'APP-INF/lib'
    deploymentDescriptor { // custom entries for application.xml:
        // fileName = "application.xml" // same as the default value
        // version = "6" // same as the default value
        applicationName = "customear"
        initializeInOrder = true
        displayName = "Custom Ear" // defaults to project.name
        // defaults to project.description if not set
        description = "My customized EAR for the Gradle documentation"
        libraryDirectory = "APP-INF/lib" // not needed, above libDirName setting dc
        module("my.jar", "java") // won't deploy as my.jar isn't deploy dependency
        webModule("my.war", "/") // won't deploy as my.war isn't deploy dependency
        securityRole "admin"
        securityRole "superadmin"
        withXml { provider -> // add a custom node to the XML
            provider.asNode().appendNode("data-source", "my/data/source")
        }
    }
}
```

You can also use customization options that the Ear task provides, such as `from` and `metaInf`.

51.8. Using custom descriptor file

You may already have appropriate settings in a `application.xml` file and want to use that instead of configuring the `ear.deploymentDescriptor` section of the build script. To accommodate that goal, place the `META-INF/application.xml` in the right place inside your source folders (see the `appDirName` property). The file contents will be used and the explicit configuration in the `ear.deploymentDescriptor` will be ignored.

52

The Jetty Plugin

This plugin is deprecated and will be removed in Gradle 4.0. Consider using the more feature-rich Gretty plugin instead.

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.

52.1. Usage

To use the Jetty plugin, include the following in your build script:

Example 52.1. Using the Jetty plugin

build.gradle

```
apply plugin: 'jetty'
```

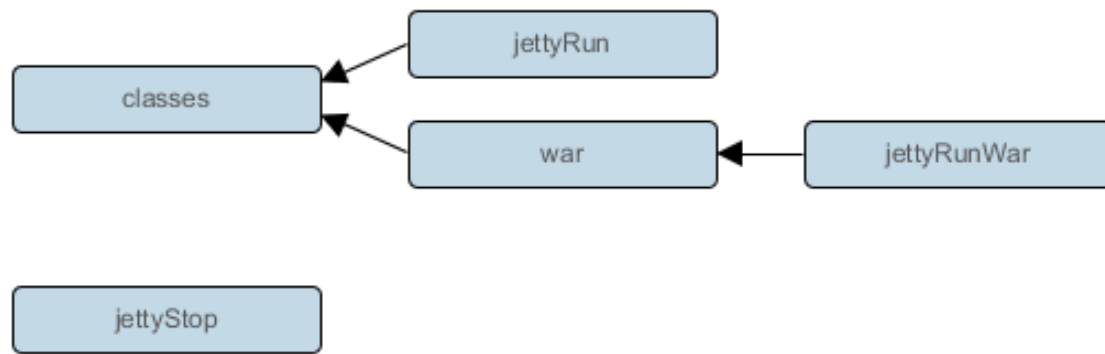
52.2. Tasks

The Jetty plugin defines the following tasks:

Table 52.1. 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 52.1. Jetty plugin - tasks



52.3. Project layout

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

52.4. Dependency management

The Jetty plugin does not define any dependency configurations.

52.5. Convention properties

The Jetty plugin defines the following convention properties:

Table 52.2. Jetty plugin - properties

Property name	Type	Default value	Description
contextPath	String	WAR file base name	The application deployment location within the Jetty container.
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.

The Application Plugin

The Application plugin facilitates creating an executable JVM application. It makes it easy to start the application locally during development, and to package the application as a TAR and/or ZIP including operating system specific start scripts.

Applying the Application plugin also implicitly applies the Java plugin. The main source set is effectively the “application”.

Applying the Application plugin also implicitly applies the Distribution plugin. A main distribution is created that packages up the application, including code dependencies and generated start scripts.

53.1. Usage

To use the application plugin, include the following in your build script:

Example 53.1. Using the application plugin

build.gradle

```
apply plugin: 'application'
```

The only mandatory configuration for the plugin is the specification of the main class (i.e. entry point) of the application.

Example 53.2. Configure the application main class

build.gradle

```
mainClassName = "org.gradle.sample.Main"
```

You can run the application by executing the **run** task (type: `JavaExec`). This will compile the main source set, and launch a new JVM with its classes (along with all runtime dependencies) as the classpath and using the specified main class. You can launch the application in debug mode with **gradle run --debug-jv** (see `JavaExec.setDebug(boolean)`).

If your application requires a specific set of JVM settings or system properties, you can configure the application property. These JVM arguments are applied to the **run** task and also considered in the generated start scripts of your distribution.

Example 53.3. Configure default JVM settings

build.gradle

```
applicationDefaultJvmArgs = ["-Dgreeting.language=en"]
```

53.1.1. The distribution

A distribution of the application can be created, by way of the Distribution plugin (which is automatically applied). A main distribution is created with the following content:

Table 53.1. Distribution content

Location	Content
(root dir)	src/dist
lib	All runtime dependencies and main source set class files.
bin	Start scripts (generated by <code>createStartScripts</code> task).

Static files to be added to the distribution can be simply added to `src/dist`. More advanced customization can be done by configuring the `CopySpec` exposed by the main distribution.

Example 53.4. Include output from other tasks in the application distribution

build.gradle

```
task createDocs {
    def docs = file("$buildDir/docs")
    outputs.dir docs
    doLast {
        docs.mkdirs()
        new File(docs, "readme.txt").write("Read me!")
    }
}

distributions {
    main {
        contents {
            from(createDocs) {
                into "docs"
            }
        }
    }
}
```

By specifying that the distribution should include the task's output files (see Section 19.9.1, “Task inputs and outputs”), Gradle knows that the task that produces the files must be invoked before the distribution can be assembled and will take care of this for you.

Example 53.5. Automatically creating files for distribution

Output of **gradle distZip**

```
> gradle distZip
:createDocs
:compileJava
:processResources NO-SOURCE
:classes
:jar
:startScripts
:distZip

BUILD SUCCESSFUL

Total time: 1 secs
```

You can run **gradle installDist** to create an image of the application in `build/install/projectName`. You can run **gradle distZip** to create a ZIP containing the distribution, **gradle distTar** to create an application TAR or **gradle assemble** to build both.

53.1.2. Customizing start script generation

The application plugin can generate Unix (suitable for Linux, Mac OS X etc.) and Windows start scripts out of the box. The start scripts launch a JVM with the specified settings defined as part of the original build and runtime environment (e.g. `JAVA_OPTS` env var). The default script templates are based on the same scripts used to launch Gradle itself, that ship as part of a Gradle distribution.

The start scripts are completely customizable. Please refer to the documentation of `CreateStartScripts` for more details and customization examples.

53.2. Tasks

The Application plugin adds the following tasks to the project.

Table 53.2. Application plugin - tasks

Task name	Depends on	Type	Description
run	classes	JavaExec	Starts the application.
startScripts	jar	CreateStartScripts	Creates OS specific scripts to run the project as a JVM application.
installDist	jar, startScripts	Sync	Installs the application into a specified directory.
distZip	jar, startScripts	Zip	Creates a full distribution ZIP archive including runtime libraries and OS specific scripts.
distTar	jar, startScripts	Tar	Creates a full distribution TAR archive including runtime libraries and OS specific scripts.

53.3. Convention properties

The application plugin adds some properties to the project, which you can use to configure its behaviour. See the `Project` class in the API documentation.

The Java Library Distribution Plugin

The Java library distribution plugin is currently incubating. Please be aware that the DSL and other configuration may change in later Gradle versions.

The Java library distribution plugin adds support for building a distribution ZIP for a Java library. The distribution contains the JAR file for the library and its dependencies.

54.1. Usage

To use the Java library distribution plugin, include the following in your build script:

Example 54.1. Using the Java library distribution plugin

build.gradle

```
apply plugin: 'java-library-distribution'
```

To define the name for the distribution you have to set the `baseName` property as shown below:

Example 54.2. Configure the distribution name

build.gradle

```
distributions {  
    main {  
        baseName = 'my-name'  
    }  
}
```

The plugin builds a distribution for your library. The distribution will package up the runtime dependencies of the library. All files stored in `src/main/dist` will be added to the root of the archive distribution. You can run “**gradle distZip**” to create a ZIP file containing the distribution.

54.2. Tasks

The Java library distribution plugin adds the following tasks to the project.

Table 54.1. Java library distribution plugin - tasks

Task name	Depends on	Type	Description
distZip	jar	Zip	Creates a full distribution ZIP archive including runtime libraries.

54.3. Including other resources in the distribution

All of the files from the `src/dist` directory are copied. To include any static files in the distribution, simply arrange them in the `src/dist` directory, or add them to the content of the distribution.

Example 54.3. Include files in the distribution

build.gradle

```
distributions {
    main {
        baseName = 'my-name'
        contents {
            from { 'src/dist' }
        }
    }
}
```

55

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 46, *Java Quickstart*.

55.1. A basic Groovy project

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

Example 55.1. Groovy plugin

build.gradle

```
apply plugin: 'groovy'
```

Note: The code for this example can be found at `samples/groovy/quickstart` in the ‘-all’ distribution of Gradle.

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 2.2.0 from the public Maven repository:

Example 55.2. Dependency on Groovy

build.gradle

```
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    compile 'org.codehaus.groovy:groovy-all:2.4.7'  
}
```

Here is our complete build file:

Example 55.3. Groovy example - complete build file

build.gradle

```
apply plugin: 'eclipse'
apply plugin: 'groovy'

repositories {
    mavenCentral()
}

dependencies {
    compile 'org.codehaus.groovy:groovy-all:2.4.7'
    testCompile 'junit:junit:4.12'
}
```

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

55.2. 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 56, *The Groovy Plugin*, and you can find more sample Groovy projects in the `samples/groovy` directory in the Gradle distribution.

The Groovy Plugin

The Groovy plugin extends the Java plugin to add support for Groovy projects. It can deal with Groovy code, mixed Groovy and Java code, and even pure Java code (although we don't necessarily recommend to use it for the latter). The plugin supports *joint compilation*, which allows you to freely mix and match Groovy and Java code, with dependencies in both directions. For example, a Groovy class can extend a Java class that in turn extends a Groovy class. This makes it possible to use the best language for the job, and to rewrite any class in the other language if needed.

56.1. Usage

To use the Groovy plugin, include the following in your build script:

Example 56.1. Using the Groovy plugin

build.gradle

```
apply plugin: 'groovy'
```

56.2. Tasks

The Groovy plugin adds the following tasks to the project.

Table 56.1. Groovy plugin - tasks

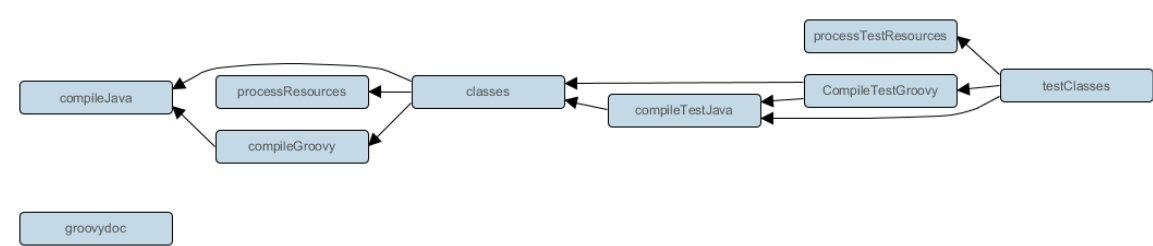
Task name	Depends on	Type	Description
compileGroovy	compileJava	GroovyCompile	Compiles production Groovy source files.
compileTestGroovy	compileTestJava	GroovyCompile	Compiles test Groovy source files.
compileSourceSetGroovy	compileSourceSetJava	GroovyCompile	Compiles the given source set's Groovy source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

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

Table 56.2. Groovy plugin - additional task dependencies

Task name	Depends on
classes	compileGroovy
testClasses	compileTestGroovy
sourceSetClasses	compileSourceSetGroovy

Figure 56.1. Groovy plugin - tasks



56.3. Project layout

The Groovy plugin assumes the project layout shown in Table 56.3, “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. ^[29] None of these directories need to exist or have anything in them; the Groovy plugin will simply compile whatever it finds.

Table 56.3. Groovy plugin - project layout

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

56.3.1. Changing the project layout

Just like the Java plugin, the Groovy plugin allows you to configure custom locations for Groovy production and test sources.

Example 56.2. Custom Groovy source layout

build.gradle

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

    test {
        groovy {
            srcDirs = ['test/groovy']
        }
    }
}
```

56.4. Dependency management

Because Gradle's build language is based on Groovy, and parts of Gradle are implemented in Groovy, Gradle already ships with a Groovy library (2.4.7 as of Gradle 3.0). Nevertheless, Groovy projects need to explicitly declare a Groovy dependency. This dependency will then be used on compile and runtime class paths. It will also be used to get hold of the Groovy compiler and Groovydoc tool, respectively.

If Groovy is used for production code, the Groovy dependency should be added to the `compile` configuration:

Example 56.3. Configuration of Groovy dependency

build.gradle

```
repositories {
    mavenCentral()
}

dependencies {
    compile 'org.codehaus.groovy:groovy-all:2.4.7'
}
```

If Groovy is only used for test code, the Groovy dependency should be added to the `testCompile` configuration:

Example 56.4. Configuration of Groovy test dependency

build.gradle

```
dependencies {
    testCompile "org.codehaus.groovy:groovy:2.4.7"
}
```

To use the Groovy library that ships with Gradle, declare a `localGroovy()` dependency. Note that different Gradle versions ship with different Groovy versions; as such, using `localGroovy()` is less safe than declaring a regular Groovy dependency.

Example 56.5. Configuration of bundled Groovy dependency

build.gradle

```
dependencies {
    compile localGroovy()
}
```

The Groovy library doesn't necessarily have to come from a remote repository. It could also come from a local `lib` directory, perhaps checked in to source control:

Example 56.6. Configuration of Groovy file dependency

build.gradle

```
repositories {
    flatDir { dirs 'lib' }
}

dependencies {
    compile module('org.codehaus.groovy:groovy:2.4.7') {
        dependency('org.ow2.asm:asm-all:5.0.3')
        dependency('antlr:antlr:2.7.7')
        dependency('commons-cli:commons-cli:1.2')
        module('org.apache.ant:ant:1.9.4') {
            dependencies('org.apache.ant:ant-junit:1.9.4@jar',
                'org.apache.ant:ant-launcher:1.9.4')
        }
    }
}
```

The “module” reference may be new to you. See Chapter 25, *Dependency Management* for more information about this and other information about dependency management.

56.5. Automatic configuration of `groovyClasspath`

The `GroovyCompile` and `Groovydoc` tasks consume Groovy code in two ways: on their `classpath`, and on their `groovyClasspath`. The former is used to locate classes referenced by the source code, and will typically contain the Groovy library along with other libraries. The latter is used to load and execute the Groovy compiler and Groovydoc tool, respectively, and should only contain the Groovy library and its dependencies.

Unless a task's `groovyClasspath` is configured explicitly, the Groovy (base) plugin will try to infer it from the task's `classpath`. This is done as follows:

- If a `groovy-all(-indy)` Jar is found on `classpath`, that jar will be added to `groovyClasspath`.
- If a `groovy(-indy)` jar is found on `classpath`, and the project has at least one repository declared, a corresponding `groovy(-indy)` repository dependency will be added to `groovyClasspath`.
- Otherwise, execution of the task will fail with a message saying that `groovyClasspath` could not be inferred.

Note that the “-indy” variation of each jar refers to the version with `invokedynamic` support.

56.6. Convention properties

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

56.7. 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.

Table 56.4. Groovy plugin - source set properties

Property name	Type	Default value	Description
<code>groovy</code>	<code>SourceDirectorySet</code> (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.
<code>groovy.srcDirs</code>	<code>Set<File></code> . Can set using anything described in Section 20.5, “Specifying a set of input 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.
<code>allGroovy</code>	<code>FileTree</code> (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 56.5. Groovy plugin - source set properties

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

56.8. GroovyCompile

The Groovy plugin adds a `GroovyCompile` task for each source set in the project. The task type extends the `JavaCompile` task (see Section 47.11, “`CompileJava`”). The `GroovyCompile` task supports most configuration options of the official Groovy compiler.

Table 56.6. Groovy plugin - GroovyCompile properties

Task Property	Type	Default Value
<code>classpath</code>	<code>FileCollection</code>	<code>sourceSet.compileClasspath</code>
<code>source</code>	<code>FileTree</code> . Can set using anything described in Section 20.5, “Specifying a set of input files”.	<code>sourceSet.groovy</code>
<code>destinationDir</code>	<code>File</code> .	<code>sourceSet.output.classesDir</code>
<code>groovyClasspath</code>	<code>FileCollection</code>	groovy configuration if non-empty; Groovy library found on classpath otherwise

56.9. Compiling and testing for Java 6

The Groovy compiler will always be executed with the same version of Java that was used to start Gradle. You should set `sourceCompatibility` and `targetCompatibility` to `1.6`. If you also have Java sources, you can follow the same steps as for the Java plugin to ensure the correct Java compiler is used.

Example 56.7. Configure Java 6 build for Groovy

gradle.properties

```
# in $HOME/.gradle/gradle.properties
java6Home=/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
```

build.gradle

```
sourceCompatibility = 1.6
targetCompatibility = 1.6

assert hasProperty('java6Home') : "Set the property 'java6Home' in your your gradle."
def javaExecutablesPath = new File(java6Home, 'bin')
def javaExecutables = [:].withDefault { execName ->
    def executable = new File(javaExecutablesPath, execName)
    assert executable.exists() : "There is no ${execName} executable in ${java6Home}"
    executable
}

tasks.withType(AbstractCompile) {
    options.with {
        fork = true
        forkOptions.executable = javaExecutables.javac
    }
}

tasks.withType(Javadoc) {
    executable = javaExecutables.javadoc
}

tasks.withType(Test) {
    executable = javaExecutables.java
}

tasks.withType(JavaExec) {
    executable = javaExecutables.java
}
```

[29] We are using the same conventions as introduced by Russel Winder's Gant tool (<https://gant.github.io/>).

57

The Scala Plugin

The Scala plugin extends the Java plugin to add support for Scala projects. It can deal with Scala code, mixed Scala and Java code, and even pure Java code (although we don't necessarily recommend to use it for the latter). The plugin supports *joint compilation*, which allows you to freely mix and match Scala and Java code, with dependencies in both directions. For example, a Scala class can extend a Java class that in turn extends a Scala class. This makes it possible to use the best language for the job, and to rewrite any class in the other language if needed.

57.1. Usage

To use the Scala plugin, include the following in your build script:

Example 57.1. Using the Scala plugin

build.gradle

```
apply plugin: 'scala'
```

57.2. Tasks

The Scala plugin adds the following tasks to the project.

Table 57.1. Scala plugin - tasks

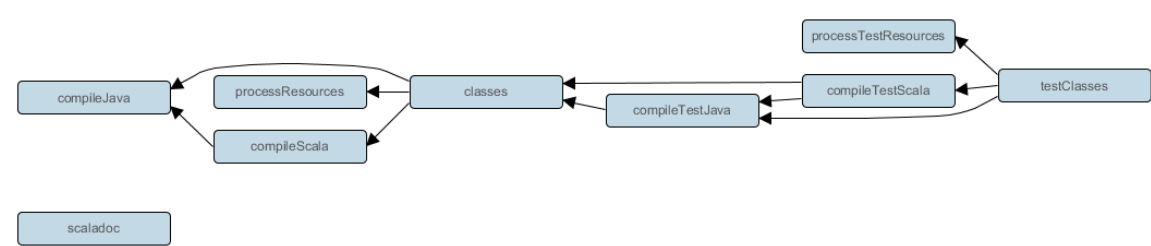
Task name	Depends on	Type	Description
compileScala	compileJava	ScalaCompile	Compiles production Scala source files.
compileTestScala	compileTestJava	ScalaCompile	Compiles test Scala source files.
compileSourceSetScala	compileSourceSetJava	ScalaCompile	Compiles the given source set's Scala source files.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

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

Table 57.2. Scala plugin - additional task dependencies

Task name	Depends on
classes	compileScala
testClasses	compileTestScala
sourceSetClasses	compileSourceSetScala

Figure 57.1. Scala plugin - tasks



57.3. 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 to exist or have anything in them; the Scala plugin will simply compile whatever it finds.

Table 57.3. Scala plugin - project layout

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

57.3.1. Changing the project layout

Just like the Java plugin, the Scala plugin allows you to configure custom locations for Scala production and test sources.

Example 57.2. Custom Scala source layout

build.gradle

```
sourceSets {
    main {
        scala {
            srcDirs = ['src/scala']
        }
    }
    test {
        scala {
            srcDirs = ['test/scala']
        }
    }
}
```

57.4. Dependency management

Scala projects need to declare a `scala-library` dependency. This dependency will then be used on compile and runtime class paths. It will also be used to get hold of the Scala compiler and Scaladoc tool, respectively. ^[30]

If Scala is used for production code, the `scala-library` dependency should be added to the `compile` configuration:

Example 57.3. Declaring a Scala dependency for production code

build.gradle

```
repositories {
    mavenCentral()
}

dependencies {
    compile 'org.scala-lang:scala-library:2.11.8'
    testCompile 'org.scalatest:scalatest_2.11:3.0.0'
    testCompile 'junit:junit:4.12'
}
```

If Scala is only used for test code, the `scala-library` dependency should be added to the `testCompile` configuration:

Example 57.4. Declaring a Scala dependency for test code

build.gradle

```
dependencies {  
    testCompile "org.scala-lang:scala-library:2.11.1"  
}
```

57.5. Automatic configuration of scalaClasspath

The `ScalaCompile` and `ScalaDoc` tasks consume Scala code in two ways: on their `classpath`, and on their `scalaClasspath`. The former is used to locate classes referenced by the source code, and will typically contain `scala-library` along with other libraries. The latter is used to load and execute the Scala compiler and Scaladoc tool, respectively, and should only contain the `scala-compiler` library and its dependencies.

Unless a task's `scalaClasspath` is configured explicitly, the Scala (base) plugin will try to infer it from the task's `classpath`. This is done as follows:

- If a `scala-library` jar is found on `classpath`, and the project has at least one repository declared, a corresponding `scala-compiler` repository dependency will be added to `scalaClasspath`.
- Otherwise, execution of the task will fail with a message saying that `scalaClasspath` could not be inferred.

57.6. Configuring the Zinc compiler

The Scala plugin uses a configuration named `zinc` to resolve the Zinc compiler and its dependencies. Gradle will provide a default version of Zinc, but if you need to use a particular Zinc version, you can add an explicit dependency like `"com.typesafe.zinc:zinc:0.3.6"` to the `zinc` configuration. Gradle supports version 0.3.0 of Zinc and above; however, due to a regression in the Zinc compiler, versions 0.3.2 through 0.3.5.2 cannot be used.

Example 57.5. Declaring a version of the Zinc compiler to use

build.gradle

```
dependencies {  
    zinc 'com.typesafe.zinc:zinc:0.3.9'  
}
```

It is important to take care when declaring your `scala-library` dependency. The Zinc compiler itself needs a compatible version of `scala-library` that may be different from the version required by your application. Gradle takes care of adding a compatible version of `scala-library` for you, but over-broad dependency resolution rules could force an incompatible version to be used instead.

For example, using `configurations.all` to force a particular version of `scala-library` would also override the version used by the Zinc compiler:

Example 57.6. Forcing a `scala-library` dependency for all configurations

build.gradle

```
configurations.all {  
    resolutionStrategy.force "org.scala-lang:scala-library:2.11.7"  
}
```

The best way to avoid this problem is to be more selective when configuring the `scala-library` dependency (such as not using a `configuration.all` rule or using a conditional to prevent the rule from being applied to the `zinc` configuration). Sometimes this rule may come from a plugin or other code that you do not have control over. In such a case, you can force a correct version of the library on the `zinc` configuration only:

Example 57.7. Forcing a `scala-library` dependency for the `zinc` configuration

build.gradle

```
configurations.zinc {  
    resolutionStrategy.force "org.scala-lang:scala-library:2.10.5"  
}
```

You can diagnose problems with the version of the Zinc compiler selected by running `dependencyInsight` for the `zinc` configuration.

57.7. Convention properties

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

57.8. 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.

Table 57.4. Scala plugin - source set properties

Property name	Type	Default value	Description
scala	SourceDirectorySet (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 Section 20.5, “Specifying a set of input files”.	[projectDir/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	FileTree (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 57.5. 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.

57.9. Compiling in external process

Scala compilation takes place in an external process.

Memory settings for the external process default to the defaults of the JVM. To adjust memory settings, configure the `scalaCompileOptions.forkOptions` property as needed:

Example 57.8. Adjusting memory settings

build.gradle

```
tasks.withType(ScalaCompile) {
    configure(scalaCompileOptions.forkOptions) {
        memoryMaximumSize = '1g'
        jvmArgs = ['-XX:MaxPermSize=512m']
    }
}
```

57.10. Incremental compilation

By compiling only classes whose source code has changed since the previous compilation, and classes affected by these changes, incremental compilation can significantly reduce Scala compilation time. It is particularly effective when frequently compiling small code increments, as is often done at development time.

The Scala plugin defaults to incremental compilation by integrating with Zinc, a standalone version of sbt's incremental Scala compiler. If you want to disable the incremental compilation, set `force = true` in your build file:

Example 57.9. Forcing all code to be compiled

build.gradle

```
tasks.withType(ScalaCompile) {
    scalaCompileOptions.with {
        force = true
    }
}
```

Note: This will only cause all classes to be recompiled if at least one input source file has changed. If there are no changes to the source files, the `compileScala` task will still be considered UP-TO-DATE as usual.

The Zinc-based Scala Compiler supports joint compilation of Java and Scala code. By default, all Java and Scala code under `src/main/scala` will participate in joint compilation. Even Java code will be compiled incrementally.

Incremental compilation requires dependency analysis of the source code. The results of this analysis are stored in the file designated by `scalaCompileOptions.incrementalOptions.analysisFile` (which has a sensible default). In a multi-project build, analysis files are passed on to downstream `ScalaCompile` tasks to enable incremental compilation across project boundaries. For `ScalaCompile` tasks added by the Scala plugin, no configuration is necessary to make this work. For other `ScalaCompile` tasks that you might add, the property `scalaCompileOptions.incrementalOptions.publishedCode` needs to be configured to point to the classes folder or Jar archive by which the code is passed on to compile class paths of downstream `ScalaCompile` tasks. Note that if `publishedCode` is not set correctly, downstream tasks may not recompile code affected by upstream changes, leading to incorrect compilation results.

Note that Zinc's Nailgun based daemon mode is not supported. Instead, we plan to enhance Gradle's own compiler daemon to stay alive across Gradle invocations, reusing the same Scala compiler. This is expected to yield another significant speedup for Scala compilation.

57.11. Compiling and testing for Java 6

The Scala compiler ignores Gradle's `targetCompatibility` and `sourceCompatibility` settings. In Scala 2.11, the Scala compiler always compiles to Java 6 compatible bytecode. In Scala 2.12, the Scala compiler always compiles to Java 8 compatible bytecode. If you also have Java sources, you can follow the same steps as for the Java plugin to ensure the correct Java compiler is used.

Example 57.10. Configure Java 6 build for Scala

gradle.properties

```
# in $HOME/.gradle/gradle.properties
java6Home=/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
```

build.gradle

```
sourceCompatibility = 1.6

assert hasProperty('java6Home') : "Set the property 'java6Home' in your your gradle."
def javaExecutablesPath = new File(java6Home, 'bin')
def javaExecutables = [:].withDefault { execName ->
    def executable = new File(javaExecutablesPath, execName)
    assert executable.exists() : "There is no ${execName} executable in ${javaExecutablesPath}"
    executable
}

tasks.withType(AbstractCompile) {
    options.with {
        fork = true
        forkOptions.executable = javaExecutables.javac
    }
}
tasks.withType(Test) {
    executable = javaExecutables.java
}
tasks.withType(JavaExec) {
    executable = javaExecutables.java
}
tasks.withType(Javadoc) {
    executable = javaExecutables.javadoc
}
```

57.12. Eclipse Integration

When the Eclipse plugin encounters a Scala project, it adds additional configuration to make the project work with Scala IDE out of the box. Specifically, the plugin adds a Scala nature and dependency container.

57.13. IntelliJ IDEA Integration

When the IDEA plugin encounters a Scala project, it adds additional configuration to make the project work with IDEA out of the box. Specifically, the plugin adds a Scala SDK (IntelliJ IDEA 14+) and a Scala compiler library that matches the Scala version on the project's class path. The Scala plugin is backwards compatible with earlier versions of IntelliJ IDEA and it is possible to add a Scala facet instead of the default Scala SDK by configuring `targetVersion` on `IdeaModel`.

Example 57.11. Explicitly specify a target IntelliJ IDEA version

build.gradle

```
idea {  
    targetVersion = "13"  
}
```

[30] See Section 57.5, “Automatic configuration of `scalaClasspath`”.

The ANTLR Plugin

The ANTLR plugin extends the Java plugin to add support for generating parsers using ANTLR.

The ANTLR plugin supports ANTLR version 2, 3 and 4.

58.1. Usage

To use the ANTLR plugin, include the following in your build script:

Example 58.1. Using the ANTLR plugin

build.gradle

```
apply plugin: 'antlr'
```

58.2. Tasks

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

Table 58.1. ANTLR plugin - tasks

Task name	Depends on	Type	Description
generateGrammarSource	-	AntlrTask	Generates the source files for all production ANTLR grammars.
generateTestGrammarSource	-	AntlrTask	Generates the source files for all test ANTLR grammars.
generateSourceSetGrammarSource		AntlrTask	Generates the source files for all ANTLR grammars for the given source set.

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

Table 58.2. ANTLR plugin - additional task dependencies

Task name	Depends on
compileJava	generateGrammarSource
compileTestJava	generateTestGrammarSource
compileSourceSetJava	generateSourceSetGrammarSource

58.3. Project layout

Table 58.3. ANTLR plugin - project layout

Directory	Meaning
src/main/antlr	Production ANTLR grammar files. If the ANTLR grammar is organized in packages, the structure in the antlr folder should reflect the package structure. This ensures that the generated sources end up in the correct target subfolder.
src/test/antlr	Test ANTLR grammar files.
src/sourceSet/antlr	ANTLR grammar files for the given source set.

58.4. Dependency management

The ANTLR plugin adds an `antlr` dependency configuration which provides the ANTLR implementation to use. The following example shows how to use ANTLR version 3.

Example 58.2. Declare ANTLR version

build.gradle

```
repositories {
    mavenCentral()
}

dependencies {
    antlr "org.antlr:antlr:3.5.2" // use ANTLR version 3
    // antlr "org.antlr:antlr4:4.5" // use ANTLR version 4
}
```

If no dependency is declared, `antlr:antlr:2.7.7` will be used as the default. To use a different ANTLR version add the appropriate dependency to the `antlr` dependency configuration as above.

58.5. Convention properties

The ANTLR plugin does not add any convention properties.

58.6. Source set properties

The ANTLR plugin adds the following properties to each source set in the project.

Table 58.4. ANTLR plugin - source set properties

Property name	Type	Default value	Description
antlr	SourceDirectorySet (read-only)	Not null	The ANTLR grammar files of this source set. Contains all .g or .g4 files found in the ANTLR source directories, and excludes all other types of files.
antlr.srcDirs	Set<File>. Can set using anything described in Section 20.5, “Specifying a set of input files”.	[projectDir/src/main/antlr4]	The source directories containing the ANTLR grammar files of this source set.

58.7. Controlling the ANTLR generator process

The ANTLR tool is executed in a forked process. This allows fine grained control over memory settings for the ANTLR process. To set the heap size of a ANTLR process, the `maxHeapSize` property of `AntlrTask` can be used. To pass additional command-line arguments, append to the `arguments` property of `AntlrTask`.

Example 58.3. setting custom max heap size and extra arguments for ANTLR

build.gradle

```
generateGrammarSource {  
    maxHeapSize = "64m"  
    arguments += ["-visitor", "-long-messages"]  
}
```


59

The Checkstyle Plugin

The Checkstyle plugin performs quality checks on your project's Java source files using Checkstyle and generates reports from these checks.

59.1. Usage

To use the Checkstyle plugin, include the following in your build script:

Example 59.1. Using the Checkstyle plugin

build.gradle

```
apply plugin: 'checkstyle'
```

The plugin adds a number of tasks to the project that perform the quality checks. You can execute the checks by running **gradle check**.

Note that Checkstyle will run with the same Java version used to run Gradle.

59.2. Tasks

The Checkstyle plugin adds the following tasks to the project:

Table 59.1. Checkstyle plugin - tasks

Task name	Depends on	Type	Description
checkstyleMain	classes	Checkstyle	Runs Checkstyle against the production Java source files.
checkstyleTest	testClasses	Checkstyle	Runs Checkstyle against the test Java source files.
checkstyleSourceSet	sourceSetClasses	Checkstyle	Runs Checkstyle against the given source set's Java source files.

The Checkstyle plugin adds the following dependencies to tasks defined by the Java plugin.

Table 59.2. Checkstyle plugin - additional task dependencies

Task name	Depends on
check	All Checkstyle tasks, including <code>checkstyleMain</code> and <code>checkstyleTest</code> .

59.3. Project layout

The Checkstyle plugin expects the following project layout:

Table 59.3. Checkstyle plugin - project layout

File	Meaning
<code>config/checkstyle/checkstyle.xml</code>	Checkstyle configuration file

59.4. Dependency management

The Checkstyle plugin adds the following dependency configurations:

Table 59.4. Checkstyle plugin - dependency configurations

Name	Meaning
<code>checkstyle</code>	The Checkstyle libraries to use

59.5. Configuration

See the `CheckstyleExtension` class in the API documentation.

59.6. Customizing the HTML report

The HTML report generated by the Checkstyle task can be customized using a XSLT stylesheet, for example to highlight specific errors or change its appearance:

Example 59.2. Customizing the HTML report

build.gradle

```
tasks.withType(Checkstyle) {
    reports {
        xml.enabled false
        html.enabled true
        html.stylesheet resources.text.fromFile('config/xsl/checkstyle-custom.xsl')
    }
}
```

View a sample Checkstyle stylesheet.

The CodeNarc Plugin

The CodeNarc plugin performs quality checks on your project's Groovy source files using CodeNarc and generates reports from these checks.

60.1. Usage

To use the CodeNarc plugin, include the following in your build script:

Example 60.1. Using the CodeNarc plugin

build.gradle

```
apply plugin: 'codenarc'
```

The plugin adds a number of tasks to the project that perform the quality checks. You can execute the checks by running **gradle check**.

60.2. Tasks

The CodeNarc plugin adds the following tasks to the project:

Table 60.1. CodeNarc plugin - tasks

Task name	Depends on	Type	Description
codenarcMain	-	CodeNarc	Runs CodeNarc against the production Groovy source files.
codenarcTest	-	CodeNarc	Runs CodeNarc against the test Groovy source files.
codenarcSourceSet		CodeNarc	Runs CodeNarc against the given source set's Groovy source files.

The CodeNarc plugin adds the following dependencies to tasks defined by the Groovy plugin.

Table 60.2. CodeNarc plugin - additional task dependencies

Task name	Depends on
check	All CodeNarc tasks, including <code>codenarcMain</code> and <code>codenarcTest</code> .

60.3. Project layout

The CodeNarc plugin expects the following project layout:

Table 60.3. CodeNarc plugin - project layout

File	Meaning
<code>config/codenarc/codenarc.xml</code>	CodeNarc configuration file

60.4. Dependency management

The CodeNarc plugin adds the following dependency configurations:

Table 60.4. CodeNarc plugin - dependency configurations

Name	Meaning
<code>codenarc</code>	The CodeNarc libraries to use

60.5. Configuration

See the `CodeNarcExtension` class in the API documentation.

The FindBugs Plugin

The FindBugs plugin performs quality checks on your project's Java source files using FindBugs and generates reports from these checks.

61.1. Usage

To use the FindBugs plugin, include the following in your build script:

Example 61.1. Using the FindBugs plugin

build.gradle

```
apply plugin: 'findbugs'
```

The plugin adds a number of tasks to the project that perform the quality checks. You can execute the checks by running **gradle check**.

Note that Findbugs will run with the same Java version used to run Gradle.

61.2. Tasks

The FindBugs plugin adds the following tasks to the project:

Table 61.1. FindBugs plugin - tasks

Task name	Depends on	Type	Description
findbugsMain	classes	FindBugs	Runs FindBugs against the production Java source files.
findbugsTest	testClasses	FindBugs	Runs FindBugs against the test Java source files.
findbugsSourceSet	sourceSetClasses	FindBugs	Runs FindBugs against the given source set's Java source files.

The FindBugs plugin adds the following dependencies to tasks defined by the Java plugin.

Table 61.2. FindBugs plugin - additional task dependencies

Task name	Depends on
check	All FindBugs tasks, including <code>findbugsMain</code> and <code>findbugsTest</code> .

61.3. Dependency management

The FindBugs plugin adds the following dependency configurations:

Table 61.3. FindBugs plugin - dependency configurations

Name	Meaning
<code>findbugs</code>	The FindBugs libraries to use

61.4. Configuration

See the `FindBugsExtension` class in the API documentation.

61.5. Customizing the HTML report

The HTML report generated by the FindBugs task can be customized using a XSLT stylesheet, for example to highlight specific errors or change its appearance:

Example 61.2. Customizing the HTML report

build.gradle

```
tasks.withType(FindBugs) {
    reports {
        xml.enabled false
        html.enabled true
        html.stylesheet resources.text.fromFile('config/xsl/findbugs-custom.xsl')
    }
}
```

View a sample FindBugs stylesheet.

The JDepend Plugin

The JDepend plugin performs quality checks on your project's source files using JDepend and generates reports from these checks.

62.1. Usage

To use the JDepend plugin, include the following in your build script:

Example 62.1. Using the JDepend plugin

build.gradle

```
apply plugin: 'jdepend'
```

The plugin adds a number of tasks to the project that perform the quality checks. You can execute the checks by running **gradle check**.

Note that JDepend will run with the same Java version used to run Gradle.

62.2. Tasks

The JDepend plugin adds the following tasks to the project:

Table 62.1. JDepend plugin - tasks

Task name	Depends on	Type	Description
jdependMain	classes	JDepend	Runs JDepend against the production Java source files.
jdependTest	testClasses	JDepend	Runs JDepend against the test Java source files.
jdependSourceSet	sourceSetClasses	JDepend	Runs JDepend against the given source set's Java source files.

The JDepend plugin adds the following dependencies to tasks defined by the Java plugin.

Table 62.2. JDepend plugin - additional task dependencies

Task name	Depends on
check	All JDepend tasks, including <code>jdependMain</code> and <code>jdependTest</code> .

62.3. Dependency management

The JDepend plugin adds the following dependency configurations:

Table 62.3. JDepend plugin - dependency configurations

Name	Meaning
<code>jdepend</code>	The JDepend libraries to use

62.4. Configuration

See the `JDependExtension` class in the API documentation.

The PMD Plugin

The PMD plugin performs quality checks on your project's Java source files using PMD and generates reports from these checks.

63.1. Usage

To use the PMD plugin, include the following in your build script:

Example 63.1. Using the PMD plugin

build.gradle

```
apply plugin: 'pmd'
```

The plugin adds a number of tasks to the project that perform the quality checks. You can execute the checks by running **gradle check**.

Note that Findbugs will run with the same Java version used to run Gradle.

63.2. Tasks

The PMD plugin adds the following tasks to the project:

Table 63.1. PMD plugin - tasks

Task name	Depends on	Type	Description
pmdMain	-	Pmd	Runs PMD against the production Java source files.
pmdTest	-	Pmd	Runs PMD against the test Java source files.
pmdSourceSet	-	Pmd	Runs PMD against the given source set's Java source files.

The PMD plugin adds the following dependencies to tasks defined by the Java plugin.

Table 63.2. PMD plugin - additional task dependencies

Task name	Depends on
check	All PMD tasks, including <code>pmdMain</code> and <code>pmdTest</code> .

63.3. Dependency management

The PMD plugin adds the following dependency configurations:

Table 63.3. PMD plugin - dependency configurations

Name	Meaning
<code>pmd</code>	The PMD libraries to use

63.4. Configuration

See the `PmdExtension` class in the API documentation.

The JaCoCo Plugin

The JaCoCo plugin is currently incubating. Please be aware that the DSL and other configuration may change in later Gradle versions.

The JaCoCo plugin provides code coverage metrics for Java code via integration with JaCoCo.

64.1. Getting Started

To get started, apply the JaCoCo plugin to the project you want to calculate code coverage for.

Example 64.1. Applying the JaCoCo plugin

build.gradle

```
apply plugin: "jacoco"
```

If the Java plugin is also applied to your project, a new task named `jacocoTestReport` is created that depends on the `test` task. The report is available at `$buildDir/reports/jacoco/test`. By default, a HTML report is generated.

64.2. Configuring the JaCoCo Plugin

The JaCoCo plugin adds a project extension named `jacoco` of type `JacocoPluginExtension`, which allows configuring defaults for JaCoCo usage in your build.

Example 64.2. Configuring JaCoCo plugin settings

build.gradle

```
jacoco {  
    toolVersion = "0.7.6.201602180812"  
    reportsDir = file("$buildDir/customJacocoReportDir")  
}
```

Table 64.1. Gradle defaults for JaCoCo properties

Property	Gradle default
reportsDir	" <i>\$buildDir/reports/jacoco</i> "

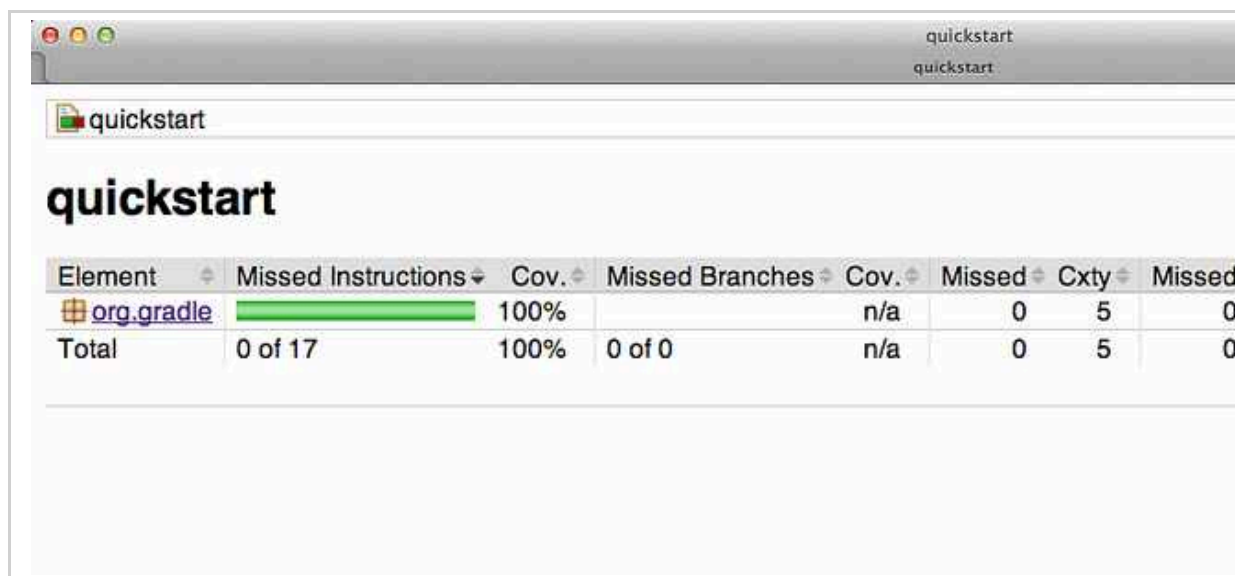
64.3. JaCoCo Report configuration

The `JacocoReport` task can be used to generate code coverage reports in different formats. It implements the standard Gradle type `Reporting` and exposes a report container of type `JacocoReportsContainer`.

Example 64.3. Configuring test task

build.gradle

```
jacocoTestReport {
    reports {
        xml.enabled false
        csv.enabled false
        html.destination "${buildDir}/jacocoHtml"
    }
}
```



64.4. Enforcing code coverage metrics

This feature requires the use of JaCoCo version 0.6.3 or higher.

The `JacocoCoverageVerification` task can be used to verify if code coverage metrics are met based on configured rules. Its API exposes the method

`JacocoCoverageVerification.violationRules(org.gradle.api.Action)` which is used as main entry point for configuring rules. Invoking any of those methods returns an instance of `JacocoViolationRulesContainer` providing extensive configuration options. The build fails if any of the configured rules are not met. JaCoCo only reports the first violated rule.

Code coverage requirements can be specified for a project as a whole, for individual files, and for particular JaCoCo-specific types of coverage, e.g., lines covered or branches covered. The following example describes the syntax.

Example 64.4. Configuring violation rules

build.gradle

```
jacocoTestCoverageVerification {
    violationRules {
        rule {
            limit {
                minimum = 0.5
            }
        }

        rule {
            enabled = false
            element = 'CLASS'
            includes = ['org.gradle.*']

            limit {
                counter = 'LINE'
                value = 'TOTALCOUNT'
                maximum = 0.3
            }
        }
    }
}
```

Note: The code for this example can be found at `samples/testing/jacoco/quickstart` in the ‘-all’ distribution of Gradle.

The `JacocoCoverageVerification` task is not a task dependency of the `check` task provided by the Java plugin. There is a good reason for it. The task is currently not incremental as it doesn't declare any outputs. Any violation of the declared rules would automatically result in a failed build when executing the `check` task. This behavior might not be desirable for all users. Future versions of Gradle might change the behavior.

64.5. JaCoCo specific task configuration

The JaCoCo plugin adds a `JacocoTaskExtension` extension to all tasks of type `Test`. This extension allows the configuration of the JaCoCo specific properties of the test task.

Example 64.5. Configuring test task

build.gradle

```
test {
    jacoco {
        append = false
        destinationFile = file("$buildDir/jacoco/jacocoTest.exec")
        classDumpDir = file("$buildDir/jacoco/classpathdumps")
    }
}
```

Table 64.2. Default values of the JaCoCo Task extension

Property	Gradle default
enabled	true
destPath	<i>\$buildDir/jacoco</i>
append	true
includes	[]
excludes	[]
excludeClassLoaders	[]
includeNoLocationClasses	false
sessionId	auto-generated
dumpOnExit	true
output	Output.FILE
address	-
port	-
classDumpPath	-
jmx	false

While all tasks of type `Test` are automatically enhanced to provide coverage information when the `java` plugin has been applied, any task that implements `JavaForkOptions` can be enhanced by the JaCoCo plugin. That is, any task that forks Java processes can be used to generate coverage information.

For example you can configure your build to generate code coverage using the `application` plugin.

Example 64.6. Using application plugin to generate code coverage data

build.gradle

```
apply plugin: "application"
apply plugin: "jacoco"

mainClassName = "org.gradle.MyMain"

jacoco {
    applyTo run
}

task applicationCodeCoverageReport(type:JacocoReport){
    executionData run
    sourceSets sourceSets.main
}
```

Note: The code for this example can be found at `samples/testing/jacoco/application` in the ‘-all’ distribution of Gradle.

Example 64.7. Coverage reports generated by applicationCodeCoverageReport

Build layout

```
application/
  build/
    jacoco/
      run.exec
      reports/jacoco/applicationCodeCoverageReport/html/
        index.html
```

64.6. Tasks

For projects that also apply the Java Plugin, The JaCoCo plugin automatically adds the following tasks:

Table 64.3. JaCoCo plugin - tasks

Task name	Depends on	Type	Description
jacocoTestReport	-	JacocoReport	Generates a code coverage report from the test task.
jacocoTestCoverageVerification	-	JacocoCoverageVerification	Verifies code coverage metrics based on specific rules for test tasks.

64.7. Dependency management

The JaCoCo plugin adds the following dependency configurations:

Table 64.4. JaCoCo plugin - dependency configurations

Name	Meaning
jacocoAnt	The JaCoCo Ant library used for running the <code>JacocoReport</code> , <code>JacocoMerge</code> and <code>JacocoCoverageVerification</code> tasks.
jacocoAgent	The JaCoCo agent library used for instrumenting the code under test.

65

The OSGi Plugin

The OSGi plugin provides a factory method to create an `OsgiManifest` object. `OsgiManifest` extends `Manifest`. To learn more about generic manifest handling, see Section 47.15.1, “Manifest”. If the Java plugins is applied, the OSGi plugin replaces the manifest object of the default jar with an `OsgiManifest` object. The replaced manifest is merged into the new one.

The OSGi plugin makes heavy use of Peter Kriens BND tool.

65.1. Usage

To use the OSGi plugin, include the following in your build script:

Example 65.1. Using the OSGi plugin

build.gradle

```
apply plugin: 'osgi'
```

65.2. Implicitly applied plugins

Applies the Java base plugin.

65.3. Tasks

This plugin does not add any tasks.

65.4. Dependency management

TBD

65.5. Convention object

The OSGi plugin adds the following convention object: `OsgiPluginConvention`

65.5.1. Convention properties

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

65.5.2. Convention methods

The OSGi plugin adds the following methods. For more details, see the API documentation of the convention object.

Table 65.1. OSGi methods

Method	Return Type	Description
<code>osgiManifest()</code>	<code>OsgiManifest</code>	Returns an <code>OsgiManifest</code> object.
<code>osgiManifest(Closure cl)</code>	<code>OsgiManifest</code>	Returns an <code>OsgiManifest</code> object configured by the closure.

The classes in the `classes` dir are analyzed regarding their 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 65.2. Configuration of OSGi MANIFEST.MF file

build.gradle

```
jar {
    manifest { // the manifest of the default jar is of type OsgiManifest
        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'
    }
}
task fooJar(type: Jar) {
    manifest = osgiManifest {
        instruction 'Bundle-Vendor', 'MyCompany'
    }
}
```

The first argument of the instruction call is the key of the property. The other arguments form the value. To learn more about the available instructions have a look at the BND tool.

The Eclipse Plugins

The Eclipse plugins generate files that are used by the Eclipse IDE, thus making it possible to import the project into Eclipse (File - Import... - Existing Projects into Workspace).

The `eclipse-wtp` is automatically applied whenever the `eclipse` plugin is applied to a War or Ear project. For utility projects (i.e. Java projects used by other web projects), you need to apply the `eclipse-wtp` plugin explicitly.

What exactly the `eclipse` plugin generates depends on which other plugins are used:

Table 66.1. Eclipse plugin behavior

Plugin	Description
None	Generates minimal <code>.project</code> file.
Java	Adds Java configuration to <code>.project</code> . Generates <code>.classpath</code> and JDT settings file.
Groovy	Adds Groovy configuration to <code>.project</code> file.
Scala	Adds Scala support to <code>.project</code> and <code>.classpath</code> files.
War	Adds web application support to <code>.project</code> file.
Ear	Adds ear application support to <code>.project</code> file.

The `eclipse-wtp` plugin generates all WTP settings files and enhances the `.project` file. If a Java or War is applied, `.classpath` will be extended to get a proper packaging structure for this utility library or web application project.

Both Eclipse plugins are open to customization and provide a standardized set of hooks for adding and removing content from the generated files.

66.1. Usage

To use either the Eclipse or the Eclipse WTP plugin, include one of the lines in your build script:

Example 66.1. Using the Eclipse plugin

build.gradle

```
apply plugin: 'eclipse'
```

Example 66.2. Using the Eclipse WTP plugin

build.gradle

```
apply plugin: 'eclipse-wtp'
```

Note: Internally, the eclipse-wtp plugin also applies the eclipse plugin so you don't need to apply both.

Both Eclipse plugins add a number of tasks to your projects. The main tasks that you will use are the `eclipse` and `cleanEclipse` tasks.

66.2. Tasks

The Eclipse plugins add the tasks shown below to a project.

Table 66.2. Eclipse plugin - tasks

Task name	Depends on	Type	Description
<code>eclipse</code>	all Eclipse configuration file generation tasks	Task	Generates all Eclipse
<code>cleanEclipse</code>	all Eclipse configuration file clean tasks	Delete	Removes all Eclipse
<code>cleanEclipseProject</code>	-	Delete	Removes the .pro
<code>cleanEclipseClasspath</code>	-	Delete	Removes the .cla
<code>cleanEclipseJdt</code>	-	Delete	Removes the .set file.
<code>eclipseProject</code>	-	GenerateEclipseProject	Generates the .pro
<code>eclipseClasspath</code>	-	GenerateEclipseClasspath	Generates the .cla
<code>eclipseJdt</code>	-	GenerateEclipseJdt	Generates the .set file.

Table 66.3. Eclipse WTP plugin - additional tasks

Task name	Depends on	Type	Description
cleanEclipseWtpComponent	-	Delete	Removes the .
cleanEclipseWtpFacet	-	Delete	Removes the . file.
eclipseWtpComponent	-	GenerateEclipseWtpComponent	Generates the .
eclipseWtpFacet	-	GenerateEclipseWtpFacet	Generates the . file.

66.3. Configuration

Table 66.4. Configuration of the Eclipse plugins

Model	Reference name	Description
EclipseModel	eclipse	Top level element that enables configuration of the Eclipse plugin in a DSL-friendly fashion.
EclipseProject	eclipse.project	Allows configuring project information
EclipseClasspath	eclipse.classpath	Allows configuring classpath information.
EclipseJdt	eclipse.jdt	Allows configuring jdt information (source/target Java compatibility).
EclipseWtpComponent	eclipse.wtp.component	Allows configuring wtp component information only if eclipse-wtp plugin was applied.
EclipseWtpFacet	eclipse.wtp.facet	Allows configuring wtp facet information only if eclipse-wtp plugin was applied.

66.4. Customizing the generated files

The Eclipse plugins allow you to customize the generated metadata files. The plugins provide a DSL for configuring model objects that model the Eclipse view of the project. These model objects are then merged with the existing Eclipse XML metadata to ultimately generate new metadata. The model objects provide lower level hooks for working with domain objects representing the file content before and after merging with the model configuration. They also provide a very low level hook for working directly with the raw XML for adjustment before it is persisted, for fine tuning and configuration that the Eclipse and Eclipse WTP plugins do not model.

66.4.1. Merging

Sections of existing Eclipse files that are also the target of generated content will be amended or overwritten, depending on the particular section. The remaining sections will be left as-is.

Disabling merging with a complete rewrite

To completely rewrite existing Eclipse files, execute a clean task together with its corresponding generation task, like “**gradle cleanEclipse eclipse**” (in that order). If you want to make this the default behavior, add “`tasks.eclipse.dependsOn(cleanEclipse)`” to your build script. This makes it unnecessary to execute the clean task explicitly.

This strategy can also be used for individual files that the plugins would generate. For instance, this can be done for the “.classpath” file with “**gradle cleanEclipseClasspath eclipseClasspath**”.

66.4.2. Hooking into the generation lifecycle

The Eclipse plugins provide objects modeling the sections of the Eclipse files that are generated by Gradle. The generation lifecycle is as follows:

1. The file is read; or a default version provided by Gradle is used if it does not exist
2. The `beforeMerged` hook is executed with a domain object representing the existing file
3. The existing content is merged with the configuration inferred from the Gradle build or defined explicitly in the eclipse DSL
4. The `whenMerged` hook is executed with a domain object representing contents of the file to be persisted
5. The `withXml` hook is executed with a raw representation of the XML that will be persisted
6. The final XML is persisted

The following table lists the domain object used for each of the Eclipse model types:

Table 66.5. Advanced configuration hooks

Model	<code>beforeMerged { arg -> }</code> argument type	<code>whenMerged { arg -> }</code> argument type	<code>with</code> argun
EclipseProject	Project	Project	XmlP
EclipseClasspath	Classpath	Classpath	XmlP
EclipseJdt	Jdt	Jdt	-
EclipseWtpComponent	WtpComponent	WtpComponent	XmlP
EclipseWtpFacet	WtpFacet	WtpFacet	XmlP

Partial overwrite of existing content

A complete overwrite causes all existing content to be discarded, thereby losing any changes made directly in the IDE. Alternatively, the `beforeMerged` hook makes it possible to overwrite just certain parts of the existing content. The following example removes all existing dependencies from the Classpath domain object:

Example 66.3. Partial Overwrite for Classpath

build.gradle

```
eclipse.classpath.file {
    beforeMerged { classpath ->
        classpath.entries.removeAll { entry -> entry.kind == 'lib' || entry.kind
    }
}
```

The resulting `.classpath` file will only contain Gradle-generated dependency entries, but not any other dependency entries that may have been present in the original file. (In the case of dependency entries, this is also the default behavior.) Other sections of the `.classpath` file will be either left as-is or merged. The same could be done for the natures in the `.project` file:

Example 66.4. Partial Overwrite for Project

build.gradle

```
eclipse.project.file.beforeMerged { project ->
    project.natures.clear()
}
```

Modifying the fully populated domain objects

The `whenMerged` hook allows to manipulate the fully populated domain objects. Often this is the preferred way to customize Eclipse files. Here is how you would export all the dependencies of an Eclipse project:

Example 66.5. Export Dependencies

build.gradle

```
eclipse.classpath.file {
    whenMerged { classpath ->
        classpath.entries.findAll { entry -> entry.kind == 'lib' }*.exported = f
    }
}
```

Modifying the XML representation

The `withXmlhook` allows to manipulate the in-memory XML representation just before the file gets written to disk. Although Groovy's XML support makes up for a lot, this approach is less convenient than manipulating the domain objects. In return, you get total control over the generated file, including sections not modeled by the domain objects.

Example 66.6. Customizing the XML

build.gradle

```
apply plugin: 'eclipse-wtp'

eclipse.wtp.facet.file.withXml { provider ->
    provider.asNode().fixed.find { it.@facet == 'jst.java' }.@facet = 'jst2.java'
}
```


67

The IDEA Plugin

The IDEA plugin generates files that are used by IntelliJ IDEA, thus making it possible to open the project from IDEA (File - Open Project). Both external dependencies (including associated source and Javadoc files) and project dependencies are considered.

What exactly the IDEA plugin generates depends on which other plugins are used:

Table 67.1. IDEA plugin behavior

Plugin	Description
None	Generates an IDEA module file. Also generates an IDEA project and workspace file if the project is the root project.
Java	Adds Java configuration to the module and project files.

One focus of the IDEA plugin is to be open to customization. The plugin provides a standardized set of hooks for adding and removing content from the generated files.

67.1. Usage

To use the IDEA plugin, include this in your build script:

Example 67.1. Using the IDEA plugin

build.gradle

```
apply plugin: 'idea'
```

The IDEA plugin adds a number of tasks to your project. The main tasks that you will use are the `idea` and `clean` tasks.

67.2. Tasks

The IDEA plugin adds the tasks shown below to a project. Notice that the `clean` task does not depend on the `cleanIdeaWorkspace` task. This is because the workspace typically contains a lot of user specific temporary data and it is not desirable to manipulate it outside IDEA.

Table 67.2. IDEA plugin - Tasks

Task name	Depends on	Type	Description
idea	ideaProject, ideaModule , ideaWorkspace		Generates all IDEA configuration files
cleanIdea	cleanIdeaProject , cleanIdeaModule	Delete	Removes all IDEA configuration files
cleanIdeaProject	-	Delete	Removes the IDEA project file
cleanIdeaModule	-	Delete	Removes the IDEA module file
cleanIdeaWorkspace	-	Delete	Removes the IDEA workspace file
ideaProject	-	GenerateIdeaProject	Generates the .ipr file. This task is only added to the root project.
ideaModule	-	GenerateIdeaModule	Generates the .iml file
ideaWorkspace	-	GenerateIdeaWorkspace	Generates the .iws file. This task is only added to the root project.

67.3. Configuration

Table 67.3. Configuration of the idea plugin

Model	Reference name	Description
IdeaModel	idea	Top level element that enables configuration of the idea plugin in a DSL-friendly fashion
IdeaProject	idea.project	Allows configuring project information
IdeaModule	idea.module	Allows configuring module information
IdeaWorkspace	idea.workspace	Allows configuring the workspace XML

67.4. Customizing the generated files

The IDEA plugin provides hooks and behavior for customizing the generated content. The workspace file can effectively only be manipulated via the `withXml` hook because its corresponding domain object is essentially empty.

The tasks recognize existing IDEA files, and merge them with the generated content.

67.4.1. Merging

Sections of existing IDEA files that are also the target of generated content will be amended or overwritten, depending on the particular section. The remaining sections will be left as-is.

Disabling merging with a complete overwrite

To completely rewrite existing IDEA files, execute a clean task together with its corresponding generation task, like “**gradle cleanIdea idea**” (in that order). If you want to make this the default behavior, add “`tasks.idea.dependsOn(cleanIdea)`” to your build script. This makes it unnecessary to execute the clean task explicitly.

This strategy can also be used for individual files that the plugin would generate. For instance, this can be done for the “`.iml`” file with “**gradle cleanIdeaModule ideaModule**”.

67.4.2. Hooking into the generation lifecycle

The plugin provides objects modeling the sections of the metadata files that are generated by Gradle. The generation lifecycle is as follows:

1. The file is read; or a default version provided by Gradle is used if it does not exist
2. The `beforeMerged` hook is executed with a domain object representing the existing file
3. The existing content is merged with the configuration inferred from the Gradle build or defined explicitly in the eclipse DSL

4. The `whenMerged` hook is executed with a domain object representing contents of the file to be persisted
5. The `withXml` hook is executed with a raw representation of the XML that will be persisted
6. The final XML is persisted

The following table lists the domain object used for each of the model types:

Table 67.4. Idea plugin hooks

Model	<code>beforeMerged { arg -> }</code> argument type	<code>whenMerged { arg -> }</code> argument type	<code>withXml { }</code> argument type
IdeaProject	Project	Project	XmlProvide
IdeaModule	Module	Module	XmlProvide
IdeaWorkspace	Workspace	Workspace	XmlProvide

Partial rewrite of existing content

A complete rewrite causes all existing content to be discarded, thereby losing any changes made directly in the IDE. The `beforeMerged` hook makes it possible to overwrite just certain parts of the existing content. The following example removes all existing dependencies from the `Module` domain object:

Example 67.2. Partial Rewrite for Module

build.gradle

```
idea.module.iml {
    beforeMerged { module ->
        module.dependencies.clear()
    }
}
```

The resulting module file will only contain Gradle-generated dependency entries, but not any other dependency entries that may have been present in the original file. (In the case of dependency entries, this is also the default behavior.) Other sections of the module file will be either left as-is or merged. The same could be done for the module paths in the project file:

Example 67.3. Partial Rewrite for Project

build.gradle

```
idea.project.ipr {
    beforeMerged { project ->
        project.modulePaths.clear()
    }
}
```

Modifying the fully populated domain objects

The `whenMerged` hook allows you to manipulate the fully populated domain objects. Often this is the preferred way to customize IDEA files. Here is how you would export all the dependencies of an IDEA module:

Example 67.4. Export Dependencies

build.gradle

```
idea.module.iml {
    whenMerged { module ->
        module.dependencies*.exported = true
    }
}
```

Modifying the XML representation

The `withXml` hook allows you to manipulate the in-memory XML representation just before the file gets written to disk. Although Groovy's XML support makes up for a lot, this approach is less convenient than manipulating the domain objects. In return, you get total control over the generated file, including sections not modeled by the domain objects.

Example 67.5. Customizing the XML

build.gradle

```
idea.project.ipr {
    withXml { provider ->
        provider.node.component
            .find { it.@name == 'VcsDirectoryMappings' }
            .mapping.@vcs = 'Git'
    }
}
```

67.5. Further things to consider

The paths of dependencies in the generated IDEA files are absolute. If you manually define a path variable pointing to the Gradle dependency cache, IDEA will automatically replace the absolute dependency paths with this path variable. you can configure this path variable via the “`idea.pathVariables`” property, so that it can do a proper merge without creating duplicates.

Part VI. The Software model

Rule based model configuration

Support for rule based configuration is currently incubating. Please be aware that the DSL, APIs and other configuration may change in later Gradle versions.

Rule based model configuration enables *configuration logic to itself have dependencies* on other elements of configuration, and to make use of the resolved states of those other elements of configuration while performing its own configuration.

68.1. Background

Rule based model configuration facilitates easier domain modelling: communicating intent (i.e. the what) over mechanics (i.e. the how). Domain modelling is a core tenet of Gradle and provides Gradle with several advantages over prior generation build tools such as Apache Ant that focus on the execution model. It allows humans to understand builds at a level that is meaningful to them.

As well as helping humans, a strong domain model also helps the dutiful machines. Plugins can more effectively collaborate around a strong domain model (e.g. plugins can say something about Java applications, such as providing conventions). Very importantly, by having a model of the *what* instead of the *how* Gradle can make intelligent choices on just how to do the how.

Gradle's support for building native software and Play Framework applications already uses this configuration model. Gradle also includes some initial support for building Java libraries using this configuration model.

68.2. Motivations for change

Domain modelling in Gradle isn't new. The Java plugin's `SourceSet` concept is an example of domain modelling, as is the modelling of `NativeBinary` in the native plugin suite.

A distinguishing characteristic of Gradle compared to other build tools that also embrace modelling is that Gradle's model is open and collaborative. Gradle is fundamentally a tool for modelling software construction and then realizing the model, via tasks such as compilation etc. Different domain plugins (e.g. Java, C++, Android) provide models that other plugins can collaborate with and build upon.

While Gradle has long employed sophisticated techniques when it comes to realizing the model (i.e. what we know as building code), the next generation of Gradle builds will employ some of the same techniques to

creation of the model itself. By defining build tasks as effectively a graph of dependent functions with explicit inputs and outputs, Gradle is able to order, cache, parallelize and apply other optimizations to the work. Using a “graph of tasks” for the production of software is a long established idea, and necessary given the complexity of software production. The task graph effectively defines the *rules* of execution that Gradle must follow. The term “Rule based model configuration” refers to applying the same concepts to building the model that builds the task graph.

Another key motivation is performance and scale. Aspects of the current approach that Gradle takes to modelling the build reduce parallelism opportunities and limit scalability. The software model is being designed with the requirements of modern software delivery in mind, where immediate responsiveness is critical for projects large and small.

68.3. Basic Concepts

68.3.1. The “model space”

The term “model space” is used to refer to the formal model, which can be read and modified by rules.

A counterpart to the model space is the “project space”, which should be familiar to readers. The “project space” is a graph of objects (e.g `project.repositories`, `project.tasks` etc.) having a `Project` as its root. A build script is effectively adding and configuring objects of this graph. For the most part, the “project space” is opaque to Gradle. It is an arbitrary graph of objects that Gradle only partially understands.

Each project also has its own model space, which is distinct from the project space. A key characteristic of the “model space” is that Gradle knows much more about it (which is knowledge that can be put to good use). The objects in the model space are “managed”, to a greater extent than objects in the project space. The origin, structure, state, collaborators and relationships of objects in the model space are first class constructs. This is effectively the characteristic that functionally distinguishes the model space from the project space: the objects of the model space are defined in ways that Gradle can understand them intimately, as opposed to an object that is the result of running relatively opaque code. A “rule” is effectively a building block of this definition.

The model space will eventually replace the project space, becoming the only “space”.

68.3.2. Rules

The model space is defined by “rules”. A rule is just a function (in the abstract sense) that either produces a model element, or acts upon a model element. Every rule has a single subject and zero or more inputs. Only the subject can be changed by a rule, while the inputs are effectively immutable.

Gradle guarantees that all inputs are fully “realized” before the rule executes. The process of “realizing” a model element is effectively executing all the rules for which it is the subject, transitioning it to its final state. There is a strong analogy here to Gradle's task graph and task execution model. Just as tasks depend on each other and Gradle ensures that dependencies are satisfied before executing a task, rules effectively depend on each other (i.e. a rule depends on all rules whose subject is one of the inputs) and Gradle ensures that all dependencies are satisfied before executing the rule.

Model elements are very often defined in terms of other model elements. For example, a compile task's

configuration can be defined in terms of the configuration of the source set that it is compiling. In this scenario, the compile task would be the subject of a rule and the source set an input. Such a rule could configure the task subject based on the source set input without concern for how it was configured, who it was configured by or when the configuration was specified.

There are several ways to declare rules, and in several forms.

68.4. Rule sources

One way to define rules is via a `RuleSource` subclass. If an object extends `RuleSource` and contains any methods annotated by '@Mutate', then each such method defines a rule. For each such method, the first argument is the subject, and zero or more subsequent arguments may follow and are inputs of the rule.

Example 68.1. applying a rule source plugin

build.gradle

```
@Managed
interface Person {
    void setFirstName(String name)
    String getFirstName()

    void setLastName(String name)
    String getLastName()
}

class PersonRules extends RuleSource {
    @Model void person(Person p) {}

    //Create a rule that modifies a Person and takes no other inputs
    @Mutate void setFirstName(Person p) {
        p.firstName = "John"
    }

    //Create a rule that modifies a ModelMap<Task> and takes as input a Person
    @Mutate void createHelloTask(ModelMap<Task> tasks, Person p) {
        tasks.create("hello") {
            doLast {
                println "Hello $p.firstName $p.lastName!"
            }
        }
    }
}

apply plugin: PersonRules
```

Output of **gradle hello**

```
> gradle hello
:hello
Hello John Smith!

BUILD SUCCESSFUL

Total time: 1 secs
```

Each of the different methods of the rule source are discrete, independent rules. Their order, or the fact that they belong to the same class, do not affect their behavior.

Example 68.2. a model creation rule

build.gradle

```
@Model void person(Person p) {}
```

This rule declares that there is a model element at path "person" (defined by the method name), of type `Person`. This is the form of the `Model` type rule for `Managed` types. Here, the person object is the rule subject. The method could potentially have a body, that mutated the person instance. It could also potentially have more parameters, which would be the rule inputs.

Example 68.3. a model mutation rule

build.gradle

```
//Create a rule that modifies a Person and takes no other inputs
@Mutate void setFirstName(Person p) {
    p.firstName = "John"
}
```

This `Mutate` rule mutates the person object. The first parameter to the method is the subject. Here, a by-type reference is used as no `Path` annotation is present on the parameter. It could also potentially have more parameters, that would be the rule inputs.

Example 68.4. creating a task

build.gradle

```
//Create a rule that modifies a ModelMap<Task> and takes as input a Person
@Mutate void createHelloTask(ModelMap<Task> tasks, Person p) {
    tasks.create("hello") {
        doLast {
            println "Hello $p.firstName $p.lastName!"
        }
    }
}
```

This `Mutate` rule effectively adds a task, by mutating the tasks collection. The subject here is the "tasks" node, which is available as a `ModelMap` of `Task`. The only input is our person element. As the person is being used as an input here, it will have been realised before executing this rule. That is, the task container effectively *depends on* the person element. If there are other configuration rules for the person element, potentially specified in a build script or other plugin, they will also be guaranteed to have been executed.

As `Person` is a `Managed` type in this example, any attempt to modify the person parameter in this method would result in an exception being thrown. `Managed` objects enforce immutability at the appropriate point in their lifecycle.

Rule source plugins can be packaged and distributed in the same manner as other types of plugins (see Chapter 41, *Writing Custom Plugins*). They also may be applied in the same manner (to project objects) as

Plugin implementations (i.e. via `Project.apply(java.util.Map)`).

Please see the documentation for `RuleSource` for more information on constraints on how rule sources must be implemented and for more types of rules.

68.5. Advanced Concepts

68.5.1. Model paths

A model path identifies the location of an element relative to the root of its model space. A common representation is a period-delimited set of names. For example, the model path `"tasks"` is the path to the element that is the task container. Assuming a task whose name is `hello`, the path `"tasks.hello"` is the path to this task.

68.5.2. Managed model elements

Currently, any kind of Java object can be part of the model space. However, there is a difference between “managed” and “unmanaged” objects.

A “managed” object is transparent and enforces immutability once realized. Being transparent means that its structure is understood by the rule infrastructure and as such each of its properties are also individual elements in the model space.

An “unmanaged” object is opaque to the the model space and does not enforce immutability. Over time, more mechanisms will be available for defining managed model elements culminating in all model elements being managed in some way.

Managed models can be defined by attaching the `@Managed` annotation to an interface:

Example 68.5. a managed type

build.gradle

```
@Managed
interface Person {
    void setFirstName(String name)
    String getFirstName()

    void setLastName(String name)
    String getLastName()
}
```

By defining a getter/setter pair, you are effectively declaring a managed property. A managed property is a property for which Gradle will enforce semantics such as immutability when a node of the model is not the subject of a rule. Therefore, this example declares properties named *firstName* and *lastName* on the managed type *Person*. These properties will only be writable when the view is mutable, that is to say when the *Person* is the subject of a `Rule` (see below the explanation for rules).

Managed properties can be of any scalar type. In addition, properties can also be of any type which is itself managed:

Property type	Nullable	Example
String	Yes	<p>Example 68.6. a String property</p> <p>build.gradle</p> <pre>void setFirstName(String name) String getFirstName()</pre>
File	Yes	<p>Example 68.7. a File property</p> <p>build.gradle</p> <pre>void setHomeDirectory(File homeDir) File getHomeDirectory()</pre>
Integer, Boolean, Byte, Short, Float, Long, Double	Yes	<p>Example 68.8. a Long property</p> <p>build.gradle</p> <pre>void setId(Long id) Long getId()</pre>
int, boolean, byte, short , float, long, double	No	<p>Example 68.9. a boolean property</p> <p>build.gradle</p> <pre>void setEmployed(boolean isEmployed) boolean isEmployed()</pre> <p>Example 68.10. an int property</p> <p>build.gradle</p> <pre>void setAge(int age) int getAge()</pre>
Another <i>managed</i> type.	Only if read/write	<p>Example 68.11. a managed property</p> <p>build.gradle</p> <pre>void setMother(Person mother) Person getMother()</pre>

An <i>enumeration</i> type.	Yes	<p>Example 68.12. an enumeration type property</p> <p>build.gradle</p> <pre>void setMaritalStatus(MaritalStatus status) MaritalStatus getMaritalStatus()</pre>
A <code>ManagedSet</code> . A managed set supports the creation of new named model elements, but not their removal.	Only if read/write	
A <code>Set</code> or <code>List</code> of scalar types. All classic operations on collections are supported: add, remove, clear...	Only if read/write	<p>Example 68.13. a managed set</p> <p>build.gradle</p> <pre>ModelSet<Person> getChildren()</pre> <p>build.gradle</p> <pre>void setUserGroups(List<String> groups) List<String> getUserGroups()</pre>

If the type of a property is itself a managed type, it is possible to declare only a getter, in which case you are declaring a read-only property. A read-only property will be instantiated by Gradle, and cannot be replaced with another object of the same type (for example calling a setter). However, the properties of that property can potentially be changed, if, and only if, the property is the subject of a rule. If it's not the case, the property is immutable, like any classic read/write managed property, and properties of the property cannot be changed at all.

Managed types can be defined out of interfaces or abstract classes and are usually defined in plugins, which are written either in Java or Groovy. Please see the `Managed` annotation for more information on creating managed model objects.

68.5.3. Model element types

There are particular types (language types) supported by the model space and can be generalised as follows:

Table 68.2. Type definitions

Type	Definition
Scalar	<p>A scalar type is one of the following:</p> <ul style="list-style-type: none"> • a primitive type (e.g. <code>int</code>) or its boxed type (e.g. <code>Integer</code>) • a <code>BigInteger</code> or <code>BigDecimal</code> • a <code>String</code> • a <code>File</code> • an enumeration type
Scalar Collection	A <code>java.util.List</code> or <code>java.util.Set</code> containing one of the scalar types
Managed type	Any class which is a valid managed model (i.e. annotated with <code>@Managed</code>)
Managed collection	A <code>ModelMap</code> or <code>ModelSet</code>

There are various contexts in which these types can be used:

Table 68.3. Model type support

Context	Supported types
Creating top level model elements	<ul style="list-style-type: none"> • Any managed type • <code>FunctionalSourceSet</code> (when the <code>LanguageBasePlugin</code> plugin has been applied) • Subtypes of <code>LanguageSourceSet</code> which have been registered via <code>ComponentType</code>
Properties of managed model elements	<p>The properties (attributes) of a managed model elements may be one or more of the following:</p> <ul style="list-style-type: none"> • A managed type • A type which is annotated with <code>@Unmanaged</code> • A Scalar Collection • A Managed collection containing managed types • A Managed collection containing <code>FunctionalSourceSet</code>'s (when the <code>LanguageBasePlugin</code> plugin has been applied) • Subtypes of <code>LanguageSourceSet</code> which have been registered via <code>ComponentType</code>

68.5.4. Language source sets

`FunctionalSourceSets` and subtypes of `LanguageSourceSet` (which have been registered via `ComponentType`) can be added to the model space via rules or via the model DSL.

Example 68.14. strongly modelling sources sets

build.gradle

```
apply plugin: 'java-lang'

//Creating LanguageSourceSets via rules
class LanguageSourceSetRules extends RuleSource {
    @Model
    void mySourceSet(JavaSourceSet javaSource) {
        javaSource.source.srcDir("src/main/my")
    }
}
apply plugin: LanguageSourceSetRules

//Creating LanguageSourceSets via the model DSL
model {
    another(JavaSourceSet) {
        source {
            srcDir "src/main/another"
        }
    }
}

//Using FunctionalSourceSets
@Managed
interface SourceBundle {
    FunctionalSourceSet getFreeSources()
    FunctionalSourceSet getPaidSources()
}
model {
    sourceBundle(SourceBundle) {
        freeSources.create("main", JavaSourceSet)
        freeSources.create("resources", JvmResourceSet)
        paidSources.create("main", JavaSourceSet)
        paidSources.create("resources", JvmResourceSet)
    }
}
```

Note: The code for this example can be found at `samples/modelRules/language-support` in the ‘-all’ distribution of Gradle.

Output of `gradle help`

```
> gradle help
:help
```

68.5.5. References, binding and scopes

As previously mentioned, a rule has a subject and zero or more inputs. The rule’s subject and inputs are declared as “references” and are “bound” to model elements before execution by Gradle. Each rule must effectively forward declare the subject and inputs as references. Precisely how this is done depends on the form of the rule. For example, the rules provided by a `RuleSource` declare references as method parameters.

A reference is either “by-path” or “by-type”.

A “by-type” reference identifies a particular model element by its type. For example, a reference to the `TaskContainer` effectively identifies the `"tasks"` element in the project model space. The model space is not exhaustively searched for candidates for by-type binding; rather, a rule is given a scope (discussed later) that determines the search space for a by-type binding.

A “by-path” reference identifies a particular model element by its path in model space. By-path references are always relative to the rule scope; there is currently no way to path “out” of the scope. All by-path references also have an associated type, but this does not influence what the reference binds to. The element identified by the path must however be type compatible with the reference, or a fatal “binding failure” will occur.

Binding scope

Rules are bound within a “scope”, which determines how references bind. Most rules are bound at the project scope (i.e. the root of the model graph for the project). However, rules can be scoped to a node within the graph. The `ModelMap.named(java.lang.String, java.lang.Class)` method is an example of a mechanism for applying scoped rules. Rules declared in the build script using the `model { }` block, or via a `RuleSource` applied as a plugin use the root of the model space as the scope. This can be considered the default scope.

By-path references are always relative to the rule scope. When the scope is the root, this effectively allows binding to any element in the graph. When it is not, then only the children of the scope can be referenced using “by-path” notation.

When binding by-type references, the following elements are considered:

- The scope element itself.
- The immediate children of the scope element.
- The immediate children of the model space (i.e. project space) root.

For the common case, where the rule is effectively scoped to the root, only the immediate children of the root need to be considered.

Binding to all elements in a scope matching type

Mutating or validating all elements of a given type in some scope is a common use-case. To accommodate this, rules can be applied via the `@Each` annotation.

In the example below, a `@Defaults` rule is applied to each `FileItem` in the model setting a default file size of `"1024"`. Another rule applies a `RuleSource` to every `DirectoryItem` that makes sure all file sizes are positive and divisible by `"16"`.

Example 68.15. a DSL example applying a rule to every element in a scope

build.gradle

```
@Managed interface Item extends Named {}
@Managed interface FileItem extends Item {
    void setSize(int size)
    int getSize()
}
@Managed interface DirectoryItem extends Item {
    ModelMap<Item> getChildren()
}

class PluginRules extends RuleSource {
    @Defaults void setDefaultFileSize(@Each FileItem file) {
        file.size = 1024
    }

    @Rules void applyValidateRules(ValidateRules rules, @Each DirectoryItem dir) {
    }
}
apply plugin: PluginRules

abstract class ValidateRules extends RuleSource {
    @Validate
    void validateSizeIsPositive(ModelMap<FileItem> files) {
        files.each { file ->
            assert file.size > 0
        }
    }

    @Validate
    void validateSizeDivisibleBySixteen(ModelMap<FileItem> files) {
        files.each { file ->
            assert file.size % 16 == 0
        }
    }
}

model {
    root(DirectoryItem) {
        children {
            dir(DirectoryItem) {
                children {
                    file1(FileItem)
                    file2(FileItem) { size = 2048 }
                }
            }
            file3(FileItem)
        }
    }
}
```

Note: The code for this example can be found at `samples/modelRules/ruleSourcePluginEach` in the ‘-all’ distribution of Gradle.

68.6. The model DSL

In addition to using a `RuleSource`, it is also possible to declare a model and rules directly in a build script using the “model DSL”.

The general form of the model DSL is:

```
model {  
    «rule-definitions»  
}
```

The model DSL makes heavy use of various Groovy DSL features. Please have a read of Section 18.7, “Some Groovy basics” for an introduction to these Groovy features.

All rules are nested inside a `model` block. There may be any number of rule definitions inside each `model` block, and there may be any number of `model` blocks in a build script. You can also use a `model` block in build scripts that are applied using `apply from: $uri`.

There are currently 2 kinds of rule that you can define using the model DSL: configuration rules, and creation rules.

68.6.1. Configuration rules

You can define a rule that configures a particular model element. A configuration rule has the following form:

```
model {  
    «model-path-to-subject» {  
        «configuration code»  
    }  
}
```

Continuing with the example so far of the model element “person” of type `Person` being present, the following DSL snippet adds a configuration rule for the person that sets its `lastName` property.

Example 68.16. DSL configuration rule

build.gradle

```
model {  
    person {  
        lastName = "Smith"  
    }  
}
```

A configuration rule specifies a path to the subject that should be configured and a closure containing the code to run when the subject is configured. The closure is executed with the subject passed as the closure delegate. Exactly what code you can provide in the closure depends on the type of the subject. This is

discussed below.

You should note that the configuration code is not executed immediately but is instead executed only when the subject is required. This is an important behaviour of model rules and allows Gradle to configure only those elements that are required for the build, which helps reduce build time. For example, let's run a task that uses the "person" object:

Example 68.17. Configuration run when required

build.gradle

```
model {  
    person {  
        println "configuring person"  
        lastName = "Smith"  
    }  
}
```

Output of **gradle showPerson**

```
> gradle showPerson  
configuring person  
:showPerson  
Hello John Smith!  
  
BUILD SUCCESSFUL  
  
Total time: 1 secs
```

You can see that before the task is run, the "person" element is configured by running the rule closure. Now let's run a task that does not require the "person" element:

Example 68.18. Configuration not run when not required

Output of **gradle somethingElse**

```
> gradle somethingElse  
:somethingElse  
Not using person  
  
BUILD SUCCESSFUL  
  
Total time: 1 secs
```

In this instance, you can see that the "person" element is not configured at all.

68.6.2. Creation rules

It is also possible to create model elements at the root level. The general form of a creation rule is:

```

model {
    «element-name»(«element-type») {
        «initialization code»
    }
}

```

The following model rule creates the "person" element:

Example 68.19. DSL creation rule

build.gradle

```

model {
    person(Person) {
        firstName = "John"
    }
}

```

A creation rule definition specifies the path of the element to create, plus its public type, represented as a Java interface or class. Only certain types of model elements can be created.

A creation rule may also provide a closure containing the initialization code to run when the element is created. The closure is executed with the element passed as the closure delegate. Exactly what code you can provide in the closure depends on the type of the subject. This is discussed below.

The initialization closure is optional and can be omitted, for example:

Example 68.20. DSL creation rule without initialization

build.gradle

```

model {
    barry(Person)
}

```

You should note that the initialization code is not executed immediately but is instead executed only when the element is required. The initialization code is executed before any configuration rules are run. For example:

Example 68.21. Initialization before configuration

build.gradle

```
model {
    person {
        println "configuring person"
        println "last name is $lastName, should be Smythe"
        lastName = "Smythe"
    }
    person(Person) {
        println "creating person"
        firstName = "John"
        lastName = "Smith"
    }
}
```

Output of **gradle showPerson**

```
> gradle showPerson
creating person
configuring person
last name is Smith, should be Smythe
:showPerson
Hello John Smythe!

BUILD SUCCESSFUL

Total time: 1 secs
```

Notice that the creation rule appears in the build script *after* the configuration rule, but its code runs before the code of the configuration rule. Gradle collects up all the rules for a particular subject before running any of them, then runs the rules in the appropriate order.

68.6.3. Model rule closures

Most DSL rules take a closure containing some code to run to configure the subject. The code you can use in this closure depends on the type of the subject of the rule.

In general, a rule closure may contain arbitrary code, mixed with some type specific DSL syntax.

ModelMap<T> subject

A ModelMap is basically a map of model elements, indexed by some name. When a ModelMap is used as the subject of a DSL rule, the rule closure can use any of the methods defined on the ModelMap interface.

A rule closure with ModelMap as a subject can also include nested creation or configuration rules. These behave in a similar way to the creation and configuration rules that appear directly under the model block.

Here is an example of a nested creation rule:

You can use the model report to determine the type of a particular model element.

Example 68.22. Nested DSL creation rule

build.gradle

```
model {  
    people {  
        john(Person) {  
            firstName = "John"  
        }  
    }  
}
```

As before, a nested creation rule defines a name and public type for the element, and optionally, a closure containing code to use to initialize the element. The code is run only when the element is required in the build.

Here is an example of a nested configuration rule:

Example 68.23. Nested DSL configuration rule

build.gradle

```
model {  
    people {  
        john {  
            lastName = "Smith"  
        }  
    }  
}
```

As before, a nested configuration rule defines the name of the element to configure and a closure containing code to use to configure the element. The code is run only when the element is required in the build.

`ModelMap` introduces several other kinds of rules. For example, you can define a rule that targets each of the elements in the map. The code in the rule closure is executed once for each element in the map, when that element is required. Let's run a task that requires all of the children of the "people" element:

Example 68.24. DSL configuration rule for each element in a map

build.gradle

```
model {
    people {
        john(Person) {
            println "creating $it"
            firstName = "John"
            lastName = "Smith"
        }
        all {
            println "configuring $it"
        }
        barry(Person) {
            println "creating $it"
            firstName = "Barry"
            lastName = "Barry"
        }
    }
}
```

Output of **gradle listPeople**

```
> gradle listPeople
creating Person 'people.barry'
configuring Person 'people.barry'
creating Person 'people.john'
configuring Person 'people.john'
:listPeople
Hello Barry Barry!
Hello John Smith!

BUILD SUCCESSFUL

Total time: 1 secs
```

Any method on `ModelMap` that accepts an `Action` as its last parameter can also be used to define a nested rule.

@Managed type subject

When a managed type is used as the subject of a DSL rule, the rule closure can use any of the methods defined on the managed type interface.

A rule closure can also configure the properties of the element using nested closures. For example:

Example 68.25. Nested DSL property configuration

build.gradle

```
model {  
    person {  
        address {  
            city = "Melbourne"  
        }  
    }  
}
```

Currently, the nested closures do not define rules and are executed immediately. Please be aware that this behaviour will change in a future Gradle release.

All other subjects

For all other types, the rule closure can use any of the methods defined by the type. There is no special DSL defined for these elements.

68.6.4. Automatic type coercion

Scalar properties in managed types can be assigned `CharSequence` values (e.g. `String`, `GString`, etc.) and they will be converted to the actual property type for you. This works for all scalar types including `File`s, which will be resolved relative to the current project.

Example 68.26. a DSL example showing type conversions

build.gradle


```

enum Temperature {
    TOO_HOT,
    TOO_COLD,
    JUST_RIGHT
}

@Managed
interface Item {
    void setName(String n); String getName()

    void setQuantity(int q); int getQuantity()

    void setPrice(float p); float getPrice()

    void setTemperature(Temperature t)
    Temperature getTemperature()

    void setDataFile(File f); File getDataFile()
}

class ItemRules extends RuleSource {
    @Model
    void item(Item item) {
        def data = item.dataFile.text.trim()
        def (name, quantity, price, temp) = data.split(',')
        item.name = name
        item.quantity = quantity
        item.price = price
        item.temperature = temp
    }

    @Defaults
    void setDefaults(Item item) {
        item.dataFile = 'data.csv'
    }

    @Mutate
    void createDataTask(ModelMap<Task> tasks, Item item) {
        tasks.create('showData') {
            doLast {
                println """
Item '$item.name'
quantity:    $item.quantity
price:       $item.price
temperature: $item.temperature"""
            }
        }
    }
}

apply plugin: ItemRules

model {
    item {
        price = "${price * (quantity < 10 ? 2 : 0.5)}"
    }
}

```

Note: The code for this example can be found at `samples/modelRules/modelDslCoercion` in the ‘-all’ distribution of Gradle.

In the above example, an `Item` is created and is initialized in `setDefault()` by providing the path to the data file. In the `item()` method the resolved `File` is parsed to extract and set the data. In the DSL block at the end, the price is adjusted based on the quantity; if there are fewer than 10 remaining the price is doubled, otherwise it is reduced by 50%. The `GString` expression is a valid value since it resolves to a `float` value in string form.

Finally, in `createDataTask()` we add the `showData` task to display all of the configured values.

68.6.5. Declaring input dependencies

Rules declared in the DSL may *depend* on other model elements through the use of a special syntax, which is of the form:

```
$.«path-to-model-element»
```

Paths are a period separated list of identifiers. To directly depend on the `firstName` of the person, the following could be used:

```
$.person.firstName
```

Example 68.27. a DSL rule using inputs

build.gradle

```
model {
    tasks {
        hello(Task) {
            def p = $.person
            doLast {
                println "Hello $p.firstName $p.lastName!"
            }
        }
    }
}
```

Note: The code for this example can be found at `samples/modelRules/modelDsl` in the ‘-all’ distribution of Gradle.

In the above snippet, the `$.person` construct is an input reference. The construct returns the value of the model element at the specified path, as its default type (i.e. the type advertised by the Model Report). It may appear anywhere in the rule that an expression may normally appear. It is not limited to the right hand side of variable assignments.

The input element is guaranteed to be fully configured before the rule executes. That is, all of the rules that mutate the element are guaranteed to have been previously executed, leaving the target element in its final, immutable, state.

Most model elements enforce immutability when being used as inputs. Any attempt to mutate such an element will result in a runtime error. However, some legacy type objects do not currently implement such checks. Regardless, it is always invalid to attempt to mutate an input to a rule.

Using `ModelMap<T>` as an input

When you use a `ModelMap` as input, each item in the map is made available as a property.

68.7. The model report

The built-in `ModelReport` task displays a hierarchical view of the elements in the model space. Each item prefixed with a `+` on the model report is a model element and the visual nesting of these elements correlates to the model path (e.g. `tasks.help`). The model report displays the following details about each model element:

Table 68.4. Model report - model element details

Detail	Description
Type	This is the underlying type of the model element and is typically a fully qualified class name.
Value	Is conditionally displayed on the report when a model element can be represented as a string.
Creator	Every model element has a creator. A creator signifies the origin of the model element (i.e. what created the model element).
Rules	Is a listing of the rules, excluding the creator rule, which are executed for a given model element. The order in which the rules are displayed reflects the order in which they are executed.

Example 68.28. model task output

Output of `gradle model`

```
> gradle model
:model

-----
Root project
-----

+ person
  | Type:      Person
  | Creator:    PersonRules#person(Person)
  | Rules:
    person { ... } @ build.gradle line 59, column 3
    PersonRules#setFirstName(Person)
+ age
  | Type:      int
  | Value:     0
```

```

        | Creator:      PersonRules#person(Person)
+ children
        | Type:        org.gradle.model.ModelSet<Person>
        | Creator:      PersonRules#person(Person)
+ employed
        | Type:        boolean
        | Value:        false
        | Creator:      PersonRules#person(Person)
+ father
        | Type:        Person
        | Value:        null
        | Creator:      PersonRules#person(Person)
+ firstName
        | Type:        java.lang.String
        | Value:        John
        | Creator:      PersonRules#person(Person)
+ homeDirectory
        | Type:        java.io.File
        | Value:        null
        | Creator:      PersonRules#person(Person)
+ id
        | Type:        java.lang.Long
        | Value:        null
        | Creator:      PersonRules#person(Person)
+ lastName
        | Type:        java.lang.String
        | Value:        Smith
        | Creator:      PersonRules#person(Person)
+ maritalStatus
        | Type:        MaritalStatus
        | Creator:      PersonRules#person(Person)
+ mother
        | Type:        Person
        | Value:        null
        | Creator:      PersonRules#person(Person)
+ userGroups
        | Type:        java.util.List<java.lang.String>
        | Value:        null
        | Creator:      PersonRules#person(Person)
+ tasks
        | Type:        org.gradle.model.ModelMap<org.gradle.api.Task>
        | Creator:      Project.<init>.tasks()
        | Rules:
            PersonRules#createHelloTask(ModelMap<Task>, Person)
+ buildEnvironment
        | Type:        org.gradle.api.tasks.diagnostics.BuildEnvironmentReportTask
        | Value:        task ':buildEnvironment'
        | Creator:      tasks.addPlaceholderAction(buildEnvironment)
        | Rules:
            copyToTaskContainer
+ components
        | Type:        org.gradle.api.reporting.components.ComponentReport
        | Value:        task ':components'
        | Creator:      tasks.addPlaceholderAction(components)
        | Rules:
            copyToTaskContainer
+ dependencies
        | Type:        org.gradle.api.tasks.diagnostics.DependencyReportTask
        | Value:        task ':dependencies'
        | Creator:      tasks.addPlaceholderAction(dependencies)
        | Rules:

```

```

        copyToTaskContainer
+ dependencyInsight
    | Type:      org.gradle.api.tasks.diagnostics.DependencyInsightReportTask
    | Value:     task ':dependencyInsight'
    | Creator:   tasks.addPlaceholderAction(dependencyInsight)
    | Rules:
        HelpTasksPlugin.Rules#addDefaultDependenciesReportConfiguration(Depend
        copyToTaskContainer
+ dependentComponents
    | Type:      org.gradle.api.reporting.dependents.DependentComponentsRepor
    | Value:     task ':dependentComponents'
    | Creator:   tasks.addPlaceholderAction(dependentComponents)
    | Rules:
        copyToTaskContainer
+ hello
    | Type:      org.gradle.api.Task
    | Value:     task ':hello'
    | Creator:   PersonRules#createHelloTask(ModelMap<Task>, Person) > creat
    | Rules:
        copyToTaskContainer
+ help
    | Type:      org.gradle.configuration.Help
    | Value:     task ':help'
    | Creator:   tasks.addPlaceholderAction(help)
    | Rules:
        copyToTaskContainer
+ init
    | Type:      org.gradle.buildinit.tasks.InitBuild
    | Value:     task ':init'
    | Creator:   tasks.addPlaceholderAction(init)
    | Rules:
        copyToTaskContainer
+ model
    | Type:      org.gradle.api.reporting.model.ModelReport
    | Value:     task ':model'
    | Creator:   tasks.addPlaceholderAction(model)
    | Rules:
        copyToTaskContainer
+ projects
    | Type:      org.gradle.api.tasks.diagnostics.ProjectReportTask
    | Value:     task ':projects'
    | Creator:   tasks.addPlaceholderAction(projects)
    | Rules:
        copyToTaskContainer
+ properties
    | Type:      org.gradle.api.tasks.diagnostics.PropertyReportTask
    | Value:     task ':properties'
    | Creator:   tasks.addPlaceholderAction(properties)
    | Rules:
        copyToTaskContainer
+ tasks
    | Type:      org.gradle.api.tasks.diagnostics.TaskReportTask
    | Value:     task ':tasks'
    | Creator:   tasks.addPlaceholderAction(tasks)
    | Rules:
        copyToTaskContainer
+ wrapper
    | Type:      org.gradle.api.tasks.wrapper.Wrapper
    | Value:     task ':wrapper'
    | Creator:   tasks.addPlaceholderAction(wrapper)

```

```
| Rules:
  copyToTaskContainer
```

68.8. Limitations and future direction

Rule based model configuration is the future of Gradle. This area is fledgling, but under very active development. Early experiments have demonstrated that this approach is more efficient, able to provide richer diagnostics and authoring assistance and is more extensible. However, there are currently many limitations.

The majority of the development to date has been focused on proving the efficacy of the approach, and building the internal rule execution engine and model graph mechanics. The user facing aspects (e.g the DSL, rule source classes) are yet to be optimized for conciseness and general usability. Likewise, many necessary configuration patterns and constructs are not yet able to be expressed via the API.

In conjunction with the addition of better syntax, a richer toolkit of configuration constructs and generally more expressive power, more tooling will be added that will enable build engineers and users alike to comprehend, modify and extend builds in new ways.

Due to the inherent nature of the rule based approach, it is more efficient at constructing the build model than today's Gradle. However, in the future Gradle will also leverage the parallelism that this approach enables both at configuration and execution time. Moreover, due to increased transparency of the model Gradle will be able to further reduce build times by caching and pre-computing the build model. Beyond improved general build performance, this will greatly improve the experience when using Gradle from tools such as IDEs.

As this area of Gradle is under active development, it will be changing rapidly. Please be sure to consult the documentation of Gradle corresponding to the version you are using and to watch for changes announced in the release notes for future versions.

Software model concepts

Support for the software model is currently incubating. Please be aware that the DSL, APIs and other configuration may change in later Gradle versions.

The software model describes how a piece of software is built and how the components of the software relate to each other. The software model is organized around some key concepts:

- A *component* is a general concept that represents some logical piece of software. Examples of components are a command-line application, a web application or a library. A component is often composed of other components. Most Gradle builds will produce at least one component.
- A *library* is a reusable component that is linked into or combined into some other component. In the Java ecosystem, a library is often built as a Jar file, and then later bundled into an application of some kind. In the native ecosystem, a library may be built as a shared library or static library, or both.
- A *source set* represents a logical group of source files. Most components are built from source sets of various languages. Some source sets contain source that is written by hand, and some source sets may contain source that is generated from something else.
- A *binary* represents some output that is built for a component. A component may produce multiple different output binaries. For example, for a C++ library, both a shared library and a static library binary may be produced. Each binary is initially configured to be built from the component sources, but additional source sets can be added to specific binary variants.
- A *variant* represents some mutually exclusive binary of a component. A library, for example, might target Java 7 and Java 8, effectively producing two distinct binaries: a Java 7 Jar and a Java 8 Jar. These are different variants of the library.
- The *API* of a library represents the artifacts and dependencies that are required to compile against that library. The API typically consists of a binary together with a set of dependencies.

Implementing model rules in a plugin

A plugin can define rules by extending `RuleSource` and adding methods that define the rules. The plugin class can either extend `RuleSource` directly or can implement `Plugin` and include a nested `RuleSource` subclass.

Refer to the API docs for `RuleSource` for more details.

70.1. Applying additional rules

A rule method annotated with `Rules` can apply a `RuleSource` to a target model element.

Building Java Libraries

Support for building Java libraries using the software model is currently incubating. Please be aware that the DSL, APIs and other configuration may change in later Gradle versions.

The Java software plugins are intended to replace the Java plugin, and leverage the Gradle software model to achieve the best performance, improved expressiveness and support for variant-aware dependency management.

71.1. Features

The Java software plugins provide:

- Support for building Java libraries and other components that run on the JVM.
- Support for several source languages.
- Support for building different variants of the same software, for different Java versions, or for any purpose.
- Build time definition and enforcement of Java library API.
- Compile avoidance.
- Dependency management between Java software components.

71.2. Java Software Model

The Java software plugins provide a *software model* that describes Java based software and how it should be built. This Java software model extends the base Gradle software model, to add support for building JVM libraries. A *JVM library* is a kind of library that is built for and runs on the JVM. It may be built from Java source, or from various other languages. All JVM libraries provide an API of some kind.

71.3. Usage

To use the Java software plugins, include the following in your build script:

Example 71.1. Using the Java software plugins

build.gradle

```
plugins {  
    id 'jvm-component'  
    id 'java-lang'  
}
```

71.4. Creating a library

A library is created by declaring a `JvmLibrarySpec` under the `components` element of the `model`:

Example 71.2. Creating a java library

build.gradle

```
model {  
    components {  
        main(JvmLibrarySpec)  
    }  
}
```

Output of **gradle build**

```
> gradle build  
:compileMainJarMainJava  
:processMainJarMainResources  
:createMainJar  
:mainApiJar  
:mainJar  
:assemble  
:check UP-TO-DATE  
:build
```

```
BUILD SUCCESSFUL
```

This example creates a library named `main`, which will implicitly create a `JavaSourceSet` named `java`. The conventions of the legacy Java plugin are observed, where Java sources are expected to be found in `src/main/java`, while resources are expected to be found in `src/main/resources`.

71.5. Source Sets

Source sets represent logical groupings of source files in a library. A library can define multiple source sets and all sources will be compiled and included in the resulting binaries. When a library is added to a build, the following source sets are added by default.

Table 71.1. Java plugin - default source sets

Source Set	Type	Directory
java	JavaSourceSet	src/\${library.name}/java
resources	JvmResourceSet	src/\${library.name}/resources

It is possible to configure an existing source set through the sources container:

Example 71.3. Configuring a source set

build.gradle

```
components {
    main {
        sources {
            java {
                // configure the "java" source set
            }
        }
    }
}
```

It is also possible to create an additional source set, using the `JavaSourceSet` type:

Example 71.4. Creating a new source set

build.gradle

```
components {
    main {
        sources {
            mySourceSet(JavaSourceSet) {
                // configure the "mySourceSet" source set
            }
        }
    }
}
```

71.6. Tasks

By default, when the plugins above are applied, no new tasks are added to the build. However, when libraries are defined, conventional tasks are added which build and package each binary of the library.

For each binary of a library, a single lifecycle task is created which executes all tasks associated with building the binary. To build all binaries, the standard **build** lifecycle task can be used.

Table 71.2. Java plugin - lifecycle tasks

Component Type	Binary Type	Lifecycle Task
JvmLibrarySpec	JvmBinarySpec	\${library.name}\${binary.name}

For each source set added to a library, tasks are added to compile or process the source files for each binary.

Table 71.3. Java plugin - source set tasks

Source Set Type	Task name	Type
JavaSourceSet	compile\${library.name}\${binary.name}\${library.name}\${sourceset.name}	PlatformJavaCompile
JvmResourceSet	process\${library.name}\${binary.name}\${library.name}\${sourceset.name}	ProcessResources

For each binary in a library, a packaging task is added to create the jar for that binary.

Table 71.4. Java plugin - packaging tasks

Binary Type	Task name	Depends on	Type	Description
JvmBinarySpec	create\${library.name}\${binary.name}	all PlatformJavaCompile and ProcessResources tasks associated with the binary	Jar	Packaging the compiled class files and resources into the binary jar

71.7. Finding out more about your project

Gradle provides a report that you can run from the command-line that shows details about the components and binaries that your project produces. To use this report, just run **gradle components**. Below is an example of running this report for one of the sample projects:

Example 71.5. The components report

Output of **gradle components**

```
> gradle components
:components
```

```
-----
Root project
-----
```

```
JVM library 'main'
-----
```

Source sets

```
Java source 'main:java'
  srcDir: src/main/java
Java source 'main:mySourceSet'
  srcDir: src/main/mySourceSet
JVM resources 'main:resources'
  srcDir: src/main/resources
```

Binaries

```
Jar 'main:jar'
  build using task: :mainJar
  target platform: java7
  tool chain: JDK 7 (1.7)
  classes dir: build/classes/main/jar
  resources dir: build/resources/main/jar
  API Jar file: build/jars/main/jar/api/main.jar
  Jar file: build/jars/main/jar/main.jar
```

Note: currently not all plugins register their components, so some components may not be visible.

```
BUILD SUCCESSFUL
```

```
Total time: 1 secs
```

71.8. Dependencies

A component in the Java software model can declare dependencies on other Java libraries. If component `main` depends on library `util`, this means that the API of `util` is required when compiling the sources of `main`, and the runtime of `util` is required when running or testing `main`. The terms 'API' and 'runtime' are examples of usages of a Java library.

71.8.1. Library usage

The 'API' usage of a Java library consists of:

- Artifact(s): the Jar file(s) containing the public classes of that library
- Dependencies: the set of other libraries that are required to compile against that library

When library `main` is compiled with a dependency on `util`, the 'API' dependencies of 'util' are resolved transitively, resulting in the complete set of libraries required to compile. For each of these libraries

(including 'util'), the 'API' artifacts will be included in the compile classpath.

Similarly, the 'runtime' usage of a Java library consists of artifacts and dependencies. When a Java component is tested or bundled into an application, the runtime usage of any runtime dependencies will be resolved transitively into the set of libraries required at runtime. The runtime artifacts of these libraries will then be included in the testing or runtime classpath.

71.8.2. Dependency types

Two types of Java library dependencies can be declared:

- Dependencies on a library defined in a local Gradle project
- Dependencies on a library published to a Maven repository

Dependencies onto libraries published to an Ivy repository are not yet supported.

71.8.3. Declaring dependencies

Dependencies may be declared for a specific `JavaSourceSet`, for an entire `JvmLibrarySpec` or as part of the `JvmApiSpec` of a component:

Example 71.6. Declaring a dependency onto a library

build.gradle

```
model {
    components {
        server(JvmLibrarySpec) {
            sources {
                java {
                    dependencies {
                        library 'core'
                    }
                }
            }
        }

        core(JvmLibrarySpec) {
            dependencies {
                library 'commons'
            }
        }

        commons(JvmLibrarySpec) {
            api {
                dependencies {
                    library 'collections'
                }
            }
        }

        collections(JvmLibrarySpec)
    }
}
```

Output of **gradle serverJar**

```
> gradle serverJar
:compileCollectionsJarCollectionsJava
:collectionsApiJar
:compileCommonsJarCommonsJava
:commonsApiJar
:compileCoreJarCoreJava
:processCoreJarCoreResources
:coreApiJar
:compileServerJarServerJava
:createServerJar
:serverApiJar
:serverJar

BUILD SUCCESSFUL
```

Dependencies declared for a source set will only be used for compiling that particular source set.

Dependencies declared for a component will be used when compiling all source sets for the component.

Dependencies declared for the component `api` are used for compiling all source sets for the component, and are also exported as part of the component's API. See [Enforcing API boundaries at compile time](#) for more details.

The previous example declares a dependency for the `java` source set of the `server` library onto the `core` library of the same project. However, it is possible to create a dependency on a library in a different project as well:

Example 71.7. Declaring a dependency onto a project with an explicit library

build.gradle

```
client(JvmLibrarySpec) {
    sources {
        java {
            dependencies {
                project ':util' library 'main'
            }
        }
    }
}
```

Output of **gradle clientJar**

```
> gradle clientJar
:util:compileMainJarMainJava
:util:mainApiJar
:compileClientJarClientJava
:clientApiJar
:createClientJar
:clientJar
```

BUILD SUCCESSFUL

When the target project defines a single library, the `library` selector can be omitted altogether:

Example 71.8. Declaring a dependency onto a project with an implicit library

build.gradle

```
dependencies {
    project ':util'
}
```

Dependencies onto libraries published to Maven repositories can be declared via `module identifiers` consisting of a `group name`, a `module name` plus an optional `version selector`:

Example 71.9. Declaring a dependency onto a library published to a Maven repository

build.gradle

```
verifier(JvmLibrarySpec) {  
    dependencies {  
        module 'asm' group 'org.ow2.asm' version '5.0.4'  
        module 'asm-analysis' group 'org.ow2.asm'  
    }  
}
```

Output of `gradle verifierJar`

```
> gradle verifierJar  
:compileVerifierJarVerifierJava  
:createVerifierJar  
:verifierApiJar  
:verifierJar  
  
BUILD SUCCESSFUL
```

A shorthand notation for module identifiers can also be used:

Example 71.10. Declaring a module dependency using shorthand notation

build.gradle

```
dependencies {  
    module 'org.ow2.asm:asm:5.0.4'  
    module 'org.ow2.asm:asm-analysis'  
}
```

Module dependencies will be resolved against the configured repositories as usual:

Example 71.11. Configuring repositories for dependency resolution

build.gradle

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

The `DependencySpecContainer` class provides a complete reference of the dependencies DSL.

71.9. Defining a Library API

Every library has an API, which consists of artifacts and dependencies that are required to compile against the library. The library may be explicitly declared for a component, or may be implied based on other component metadata.

By default, all `public` types of a library are considered to be part of its API. In many cases this is not ideal; a library will contain many public types that intended for internal use within that library. By explicitly declaring an API for a Java library, Gradle can provide compile-time encapsulation of these

internal-but-public types. The types to include in a library API are declared at the package level. Packages containing API types are considered to be *exported*.

By default, dependencies of a library are *not* considered to be part of its API. By explicitly declaring a dependency as part of the library API, this dependency will then be made available to consumers when compiling. Dependencies declared this way are considered to be *exported*, and are known as 'API dependencies'.

JDK 9 will introduce *Jigsaw*, the reference implementation of the *Java Module System*. Jigsaw will provide both compile-time and run-time enforcement of API encapsulation.

Gradle anticipates the arrival of JDK 9 and the Java Module System with an approach to specifying and enforcing API encapsulation at compile-time. This allows Gradle users to leverage the many benefits of strong encapsulation, and prepare their software projects for migration to JDK 9.

71.9.1. Some terminology

- An *API* is a set of classes, interfaces, methods that are exposed to a consumer.
- An *API specification* is the specification of classes, interfaces or methods that belong to an API, together with the set of dependencies that are part of the API. It can be found in various forms, like `module-info.java` in Jigsaw, or the `api { ... }` block that Gradle defines as part of those stories. Usually, we can simplify this to a list of packages, called *exported packages*.
- A *runtime jar* consists of *API classes* and *non-API classes* used at execution time. There can be multiple runtime jars depending on combinations of the variant dimensions: target platform, hardware infrastructure, target application server, ...
- *API classes* are classes of a *variant* which match the *API specification*.
- *Non-API classes* are classes of a *variant* which do not match the *API specification*.
- A *stubbed API class* is an *API class* for which its implementation and non public members have been removed. It is meant to be used when a consumer is going to be compiled against an *API*.
- An *API jar* is a collection of *API classes*. There can be multiple API jars depending on the combinations of variant dimensions.
- A *stubbed API jar* is a collection of *stubbed API classes*. There can be multiple stubbed API jars depending on the combinations of variant dimensions.
- An *ABI (application binary interface)* corresponds to the public signature of an API, that is to say the set of stubbed API classes that it exposes (and their API visible members).

We avoid the use of the term *implementation* because it is too vague: both *API classes* and *Non-API classes* can have an implementation. For example, an *API class* can be an interface, but also a concrete class. Implementation is an overloaded term in the Java ecosystem, and often refers to a class implementing an interface. This is not the case here: a concrete class can be member of an API, but to compile against an API, you don't need the implementation of the class: all you need is the signatures.

71.9.2. Specifying API classes

Example 71.12. Specifying api packages

build.gradle

```
model {
    components {
        main(JvmLibrarySpec) {
            api {
                exports 'org.gradle'
                exports 'org.gradle.utils'
            }
        }
    }
}
```

71.9.3. Specifying API dependencies

Example 71.13. Specifying api dependencies

build.gradle

```
commons(JvmLibrarySpec) {
    api {
        dependencies {
            library 'collections'
        }
    }
}
```

71.9.4. Compile avoidance

When you define an API for your library, Gradle enforces the usage of that API at compile-time. This comes with 3 direct consequences:

- Trying to use a non-API class in a dependency will now result in a compilation error.
- Changing the implementation of an API class will not result in recompilation of consumers if the ABI doesn't change (that is to say, all public methods have the same signature but not necessarily the same body).
- Changing the implementation of a non-API class will not result in recompilation of consumers. This means that changes to non-API classes will not trigger recompilation of downstream dependencies, because the ABI of the component doesn't change.

Given a main component that exports `org.gradle`, `org.gradle.utils` and defines those classes:

Example 71.14. Main sources

src/main/java/org/gradle/Person.java

```
package org.gradle;

public class Person {
    private final String name;

    public Person(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

src/main/java/org/gradle/internal/PersonInternal.java

```
package org.gradle.internal;

import org.gradle.Person;

public class PersonInternal extends Person {
    public PersonInternal(String name) {
        super(name);
    }
}
```

src/main/java/org/gradle/utils/StringUtils.java

```
package org.gradle.utils;

public abstract class StringUtils {
}
```

Compiling a component client that declares a dependency onto main will succeed:

Example 71.15. Client component

build.gradle

```
model {
    components {
        client(JvmLibrarySpec) {
            sources {
                java {
                    dependencies {
                        library 'main'
                    }
                }
            }
        }
    }
}
```

src/client/java/org/gradle/Client.java

```
package org.gradle;

public class Client {
    private Person person;

    public void setPerson(Person p) { this.person = p; }
    public Person getPerson() { return person; }
}
```

Output of **gradle :clientJar**

```
> gradle :clientJar
:compileMainJarMainJava
:processMainJarMainResources
:mainApiJar
:compileClientJarClientJava
:clientApiJar
:createClientJar
:clientJar

BUILD SUCCESSFUL
```

But trying to compile a component *brokenclient* that declares a dependency onto *main* but uses an non-API class of *main* will result in a compile-time error:

Example 71.16. Broken client component

src/brokenclient/java/org/gradle/Client.java

```
package org.gradle;

import org.gradle.internal.PersonInternal;

public class Client {
    private PersonInternal person;

    public void setPerson(PersonInternal p) { this.person = p; }
    public PersonInternal getPerson() { return person; }
}
```

Output of **gradle :brokenclientJar**

```
> gradle :brokenclientJar
:compileMainJarMainJava
:processMainJarMainResources
:mainApiJar
:compileBrokenclientJarBrokenclientJava FAILED
```

BUILD FAILED

On the other hand, if *Person.java* in *client* is updated and its API hasn't changed, *client* will not be recompiled. This is in particular important for incremental builds of large projects, where we can avoid the compilation of dependencies in chain, and then dramatically reduce build duration:

Example 71.17. Recompiling the client

src/main/java/org/gradle/Person.java

```
package org.gradle;

public class Person {
    private final String name;

    public Person(String name) {
        // we updated the body if this method
        // but the signature doesn't change
        // so we will not recompile components
        // that depend on this class
        this.name = name.toUpperCase();
    }

    public String getName() {
        return name;
    }
}
```

Output of **gradle :clientJar**

```
> gradle :clientJar
:compileMainJarMainJava
:processMainJarMainResources UP-TO-DATE
:mainApiJar
:compileClientJarClientJava UP-TO-DATE
:clientApiJar UP-TO-DATE
:createClientJar UP-TO-DATE
:clientJar UP-TO-DATE
```

BUILD SUCCESSFUL

71.10. Platform aware dependency management

71.10.1. Specifying the target platform

The software model extracts the target platform as a core concept. In the Java world, this means that a library can be built, or resolved, against a specific version of Java. For example, if you compile a library for Java 5, we know that such a library can be consumed by a library built for Java 6, but the opposite is not true. Gradle lets you define which platforms a library targets, and will take care of:

- generating a binary for each target platform (eg, a Java 5 jar as well as a Java 6 jar)
- resolving dependencies against a matching platform

The `targetPlatform` DSL defines which platforms a library should be built against:

Example 71.18. Declaring target platforms

core/build.gradle

```
model {
    components {
        main(JvmLibrarySpec) {
            targetPlatform 'java5'
            targetPlatform 'java6'
        }
    }
}
```

Output of **gradle :core:build**

```
> gradle :core:build
:core:compileMainJava5JarMainJava
:core:processMainJava5JarMainResources
:core:createMainJava5Jar
:core:mainJava5ApiJar
:core:mainJava5Jar
:core:compileMainJava6JarMainJava
:core:compileMainJava6JarMainJava6JarJava
:core:processMainJava6JarMainResources
:core:createMainJava6Jar
:core:mainJava6ApiJar
:core:mainJava6Jar
:core:assemble
:core:check UP-TO-DATE
:core:build
```

BUILD SUCCESSFUL

When building the application, Gradle generates two binaries: `java5MainJar` and `java6MainJar` corresponding to the target versions of Java. These artifacts will participate in dependency resolution as described here.

71.10.2. Binary specific source sets

For each `JvmLibrarySpec` it is possible to define additional source sets for each binary. A common use case for this is having specific dependencies for each variant and source sets that conform to those dependencies. The example below configures a `java6` source set on the `main.java6Jar` binary:

Example 71.19. Declaring binary specific sources

core/build.gradle

```
main {
    binaries.java6Jar {
        sources {
            java(JavaSourceSet) {
                source.srcDir 'src/main/java6'
            }
        }
    }
}
```

Output of **gradle clean :core:mainJava6Jar**

```
> gradle clean :core:mainJava6Jar
:core:clean UP-TO-DATE
:server:clean UP-TO-DATE
:core:compileMainJava6JarMainJava
:core:compileMainJava6JarMainJava6JarJava
:core:processMainJava6JarMainResources
:core:createMainJava6Jar
:core:mainJava6ApiJar
:core:mainJava6Jar
```

BUILD SUCCESSFUL

71.10.3. Dependency resolution

When a library targets multiple versions of Java and depends on another library, Gradle will make its best effort to resolve the dependency to the most appropriate version of the dependency library. In practice, this means that Gradle chooses the *highest compatible* version:

- for a binary B built for Java n
- for a dependency binary D built for Java m
- D is compatible with B if $m \leq n$
- for multiple compatible binaries $D(\text{java } 5)$, $D(\text{java } 6)$, $\dots D(\text{java } m)$, choose the compatible D binary with the highest Java version

Example 71.20. Declaring target platforms

server/build.gradle

```
model {
    components {
        main(JvmLibrarySpec) {
            targetPlatform 'java5'
            targetPlatform 'java6'
            sources {
                java {
                    dependencies {
                        project ':core' library 'main'
                    }
                }
            }
        }
    }
}
```

Output of **gradle clean :server:build**

```
> gradle clean :server:build
:core:clean UP-TO-DATE
:server:clean UP-TO-DATE
:core:compileMainJava5JarMainJava
:core:processMainJava5JarMainResources
:core:mainJava5ApiJar
:server:compileMainJava5JarMainJava
:server:createMainJava5Jar
:server:mainJava5ApiJar
:server:mainJava5Jar
:core:compileMainJava6JarMainJava
:core:compileMainJava6JarMainJava6JarJava
:core:processMainJava6JarMainResources
:core:mainJava6ApiJar
:server:compileMainJava6JarMainJava
:server:createMainJava6Jar
:server:mainJava6ApiJar
:server:mainJava6Jar
:server:assemble
:server:check UP-TO-DATE
:server:build
```

BUILD SUCCESSFUL

In the example above, Gradle automatically chooses the Java 6 variant of the dependency for the Java 6 variant of the `server` component, and chooses the Java 5 version of the dependency for the Java 5 variant of the `server` component.

71.11. Custom variant resolution

The Java plugin, in addition to the target platform resolution, supports resolution of custom variants. Custom variants can be defined on custom binary types, as long as they extend `JarBinarySpec`. Users interested in testing this incubating feature can check out the documentation of the `Variant` annotation.

71.12. Testing Java libraries

71.12.1. Standalone JUnit test suites

The Java software model supports defining standalone JUnit test suites as components of the model. Standalone test suite are components that are self contained, in the sense that there is no component under test: everything being tested must belong to the test suite sources.

A test suite is declared by creating a component of type `JUnitTestSuiteSpec`, which is available when you apply the `junit-test-suite` plugin:

Example 71.21. Using the JUnit plugin

build.gradle

```
plugins {  
    id 'jvm-component'  
    id 'java-lang'  
    id 'junit-test-suite'  
}  
  
model {  
    testSuites {  
        test(JUnitTestSuiteSpec) {  
            junitVersion '4.12'  
        }  
    }  
}
```

In the example above, `test` is the name of our test suite. By convention, Gradle will create two source sets for the test suite, based on the name of the component: one for Java sources, and the other for resources: `src/test/java` and `src/test/resources`. If the component was named `integTest`, then sources and resources would have been found respectively in `src/integTest/java` and `src/integTest/resources`.

Once the component is created, the test suite can be executed running the `<<test suite name>>BinaryTest` task:

Example 71.22. Executing the test suite

src/test/java/org/gradle/MyTest.java

```
package org.gradle;

import org.junit.Test;

import static org.junit.Assert.*;

public class MyTest {
    @Test
    public void myTestMethod() {
        assertEquals(4, "test".length());
    }
}
```

Output of **gradle testBinaryTest**

```
> gradle testBinaryTest
:compileTestBinaryTestJava
:processTestBinaryTestResources
:testBinaryTest

BUILD SUCCESSFUL
```

It is possible to configure source sets in a similar way as libraries.

A test suite being a component can also declare dependencies onto other components.

A test suite can also contain resources, in which case it is possible to configure the resource processing task:

Example 71.23. Executing the test suite

build.gradle

```
model {
    tasks.processTestBinaryTestResources {
        // uncomment lines
        filter { String line ->
            line.replaceAll('<!-- (.+?) -->', '$1')
        }
    }
}
```

71.12.2. Testing JVM libraries with JUnit

It is likely that you will want to test another JVM component. The Java software model supports it exactly like standalone test suites, by just declaring an additional component under test:

Example 71.24. Declaring a component under test

build.gradle

```
model {
    components {
        main(JvmLibrarySpec)
    }
    testSuites {
        test(JUnitTestSuiteSpec) {
            junitVersion '4.12'
            testing $.components.main
        }
    }
}
```

Output of **gradle testMainJarBinaryTest**

```
> gradle testMainJarBinaryTest
:compileMainJarMainJava
:processMainJarMainResources
:compileTestMainJarBinaryTestJava
:testMainJarBinaryTest
```

BUILD SUCCESSFUL

Note that the syntax to choose the component under test is a reference (`$.`). You can select any `JvmComponentSpec` as the component under test. It's also worth noting that when you declare a component under test, a test suite is created for each binary of the component under test (for example, if the component under test has a Java 7 and Java 8 version, 2 different test suite binaries would be automatically created).

71.13. Declaring Java toolchains

You can declare the list of local JVM installations using the `javaInstallations` model block. Gradle will use this information to locate your JVMs and probe their versions. Please note that this information is not yet used by Gradle to select the appropriate JDK or JRE when compiling your Java sources, or when executing Java applications. A local Java installation can be declared using the `LocalJava` type, independently of the fact they are a JDK or a JRE:

Example 71.25. Declaring local Java installations

build.gradle

```
model {
    javaInstallations {
        openJdk6(LocalJava) {
            path '/usr/lib/jvm/jdk1.6.0-amd64'
        }
        oracleJre7(LocalJava) {
            path '/usr/lib/jvm/jre1.7.0'
        }
        ibmJdk8(LocalJava) {
            path '/usr/lib/jvm/jdk1.8.0'
        }
    }
}
```

Building Play applications

Support for building Play applications is currently incubating. Please be aware that the DSL, APIs and other configuration may change in later Gradle versions.

Play is a modern web application framework. The Play plugin adds support for building, testing and running Play applications with Gradle.

The Play plugin makes use of the Gradle software model.

72.1. Usage

To use the Play plugin, include the following in your build script to apply the `play` plugin and add the Typesafe repositories:

Example 72.1. Using the Play plugin

build.gradle

```
plugins {  
    id 'play'  
}  
  
repositories {  
    jcenter()  
    maven {  
        name "typesafe-maven-release"  
        url "https://repo.typesafe.com/typesafe/maven-releases"  
    }  
    ivy {  
        name "typesafe-ivy-release"  
        url "https://repo.typesafe.com/typesafe/ivy-releases"  
        layout "ivy"  
    }  
}
```

Note that defining the Typesafe repositories is necessary. In future versions of Gradle, this will be replaced with a more convenient syntax.

72.2. Limitations

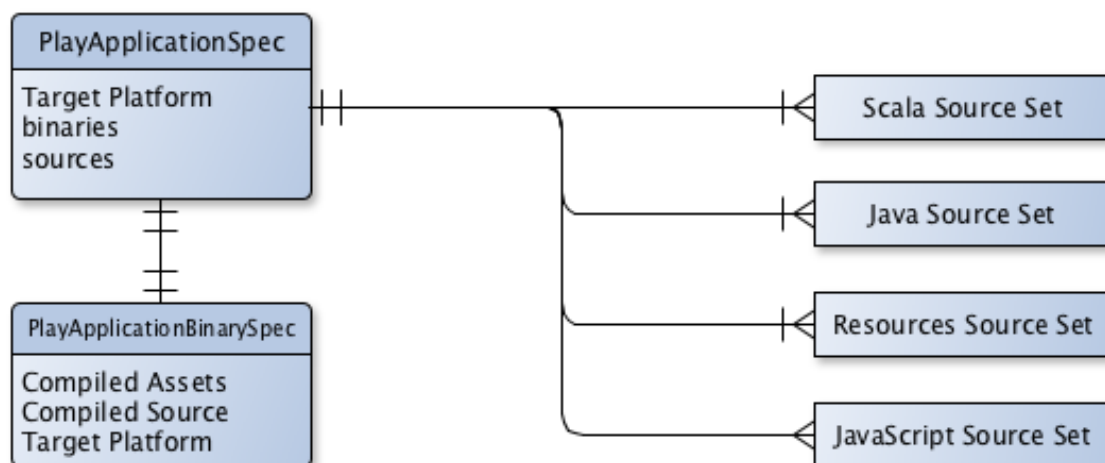
The Play plugin currently has a few limitations.

- Full support is limited to Play 2.3.x applications. Limited support is available for Play 2.4.x & 2.5.x applications. Gradle does not include support for a few new build-related features in 2.4.0+ Specifically, Gradle does not yet support aggregate reverse routes. Future Gradle versions will add more support for Play 2.6.x.
- A given project may only define a single Play application. This means that a single project cannot build more than one Play application. However, a multi-project build can have many projects that each define their own Play application.
- Play applications can only target a single “platform” (combination of Play, Scala and Java version) at a time. This means that it is currently not possible to define multiple variants of a Play application that, for example, produce jars for both Scala 2.10 and 2.11. This limitation may be lifted in future Gradle versions.
- Support for generating IDE configurations for Play applications is limited to IDEA.

72.3. Software Model

The Play plugin uses a *software model* to describe a Play application and how to build it. The Play software model extends the base Gradle software model to add support for building Play applications. A Play application is represented by a `PlayApplicationSpec` component type. The plugin automatically creates a single `PlayApplicationBinarySpec` instance when it is applied. Additional Play components cannot be added to a project.

Figure 72.1. Play plugin - software model



72.3.1. The Play application component

A Play application component describes the application to be built and consists of several configuration elements. One type of element that describes the application are the source sets that define where the application controller, route, template and model class source files should be found. These source sets are logical groupings of files of a particular type and a default source set for each type is created when the `play` plugin is applied.

Table 72.1. Default Play source sets

Source Set	Type	Directory	Filters
java	JavaSourceSet	app	**/*.java
scala	ScalaLanguageSourceSet	app	**/*.scala
routes	RoutesSourceSet	conf	routes, *.routes
twirlTemplates	TwirlSourceSet	app	**/*.html
javascript	JavaScriptSourceSet	app/assets	**/*.js

These source sets can be configured or additional source sets can be added to the Play component. See [Configuring](#) for further information.

Another element of configuring a Play application is the [*platform*](#). To build a Play application, Gradle needs to understand which versions of Play, Scala and Java to use. The Play component specifies this requirement as a `PlayPlatform`. If these values are not configured, a default version of Play, Scala and Java will be used. See [Targeting a certain version of Play](#) for information on configuring the Play platform.

Note that only a single platform can be specified for a given Play component. This means that only a single version of Play, Scala and Java can be used to build a Play component. In other words, a Play component can only produce one set of outputs, and those outputs will be built using the versions specified by the platform configured on the component.

72.3.2. The Play application binary

A Play application component is compiled and packaged to produce a set of outputs which are represented by a `PlayApplicationBinarySpec`. The Play binary specifies the jar files produced by building the component as well as providing elements by which additional content can be added to those jar files. It also exposes the tasks involved in building the component and creating the binary.

See [Configuring Play](#) for examples of configuring the Play binary.

72.4. Project Layout

The Play plugin follows the typical Play application layout. You can configure source sets to include additional directories or change the defaults.

app	Application source code.
assets	Assets that require compilation.
javascripts	JavaScript source code to be minified.
controllers	Application controller source code.
models	Application business source code.
views	Application UI templates.
build.gradle	Your project's build script.
conf	Main application configuration file and routes files.
public	Public assets.
images	Application image files.
javascripts	Typically JavaScript source code.
stylesheets	Typically CSS source code.
test	Test source code.

72.5. Tasks

The Play plugin hooks into the normal Gradle lifecycle tasks such as assemble, check and build, but it also adds several additional tasks which form the lifecycle of a Play project:

Table 72.2. Play plugin - lifecycle tasks

Task name	Depends on	Type	Description
playBinary	All compile tasks for source sets added to the Play application.	Task	Performs a build of just the Play application.
dist	createPlayBinaryZipDist, createPlayBinaryDist	Task	Assembles the Play distribution.
stage	stagePlayBinaryDist	Task	Stages the Play distribution.

The plugin also provides tasks for running, testing and packaging your Play application:

Table 72.3. Play plugin - running and testing tasks

Task name	Depends on	Type	Description
runPlayBinary	playBinary to build Play application.	PlayRun	Runs the Play application for local development. See how this works w
testPlayBinary	playBinary to build Play application and compilePlayBinaryTests	Test	Runs JUnit/TestNG tests for the Play application.

For the different types of sources in a Play application, the plugin adds the following compilation tasks:

Table 72.4. Play plugin - source set tasks

Task name	Source Type	Type	Description
<code>compilePlayBinaryScala</code>	Scala and Java	<code>PlatformScalaCompile</code>	Compile all Scala and Java sources defined in the Play application.
<code>compilePlayBinaryPlayTwirlTemplates</code>	Twirl HTML templates	<code>TwirlCompile</code>	Compile Twirl HTML templates with the Twirl compiler.
<code>compilePlayBinaryPlayRoutes</code>	Play Route files	<code>RoutesCompile</code>	Compile routes files into Scala sources.
<code>minifyPlayBinaryJavaScript</code>	JavaScript files	<code>JavaScriptMinify</code>	Minify JavaScript files with the GooG Closure compiler.

72.6. Finding out more about your project

Gradle provides a report that you can run from the command-line that shows some details about the components and binaries that your project produces. To use this report, just run **gradle components**. Below is an example of running this report for one of the sample projects:

Example 72.2. The components report

Output of **gradle components**

```
> gradle components
:components
```

```
-----
Root project
-----
```

```
Play Application 'play'
-----
```

Source sets

```
Java source 'play:java'
  srcDir: app
  includes: **/*.java
JavaScript source 'play:javascript'
  srcDir: app/assets
  includes: **/*.js
JVM resources 'play:resources'
  srcDir: conf
Routes source 'play:routes'
  srcDir: conf
  includes: routes, *.routes
Scala source 'play:scala'
  srcDir: app
  includes: **/*.scala
Twirl template source 'play:twirlTemplates'
  srcDir: app
  includes: **/*.html
```

Binaries

```
Play Application Jar 'play:binary'
  build using task: :playBinary
  target platform: Play Platform (Play 2.3.9, Scala: 2.11, Java: Java SE 8)
  toolchain: Default Play Toolchain
  classes dir: build/playBinary/classes
  resources dir: build/playBinary/resources
  JAR file: build/playBinary/lib/basic.jar
```

Note: currently not all plugins register their components, so some components may nc

BUILD SUCCESSFUL

Total time: 1 secs

72.7. Running a Play application

The `runPlayBinary` task starts the Play application under development. During development it is beneficial to execute this task as a continuous build. Continuous build is a generic feature that supports automatically re-running a build when inputs change. The `runPlayBinary` task is “continuous build aware” in that it behaves differently when run as part of a continuous build.

When not run as part of a continuous build, the `runPlayBinary` task will *block* the build. That is, the

task will not complete as long as the application is running. When running as part of a continuous build, the task will start the application if not running and otherwise propagate any changes to the code of the application to the running instance. This is useful for quickly iterating on your Play application with an edit->rebuild->refresh cycle. Changes to your application will not take affect until the end of the overall build.

To enable continuous build, run Gradle with `-t runPlayBinary` or `--continuous runPlayBinary`.

Users of Play used to such a workflow with Play's default build system should note that compile errors are handled differently. If a build failure occurs during a continuous build, the Play application will not be reloaded. Instead, you will be presented with an exception message. The exception message will only contain the overall cause of the build failure. More detailed information will only be available from the console.

72.8. Configuring a Play application

72.8.1. Targeting a certain version of Play

By default, Gradle uses Play 2.3.9, Scala 2.11 and the version of Java used to start the build. A Play application can select a different version by specifying a target `PlayApplicationSpec.platform(java.lang.Object)` on the Play application component.

Example 72.3. Selecting a version of the Play Framework

build.gradle

```
model {
    components {
        play {
            platform play: '2.3.6', scala: '2.10'
        }
    }
}
```

72.8.2. Adding dependencies

You can add compile, test and runtime dependencies to a Play application through `Configuration` created by the Play plugin.

If you are coming from SBT, the Play SBT plugin provides short names for common dependencies. For instance, if your project has a dependency on `ws`, you will need to add a dependency to `com.typesafe.play:where 2.11 is your Scala version and 2.3.9 is your Play framework version.`

Other dependencies that have short names, such as `jacksons` may actually be multiple dependencies. For those dependencies, you will need to work out the dependency coordinates from a dependency report.

- `play` is used for compile time dependencies.
- `playTest` is used for test compile time dependencies.
- `playRun` is used for run time dependencies.

Example 72.4. Adding dependencies to a Play application

build.gradle

```
dependencies {  
    play "commons-lang:commons-lang:2.6"  
}
```

72.8.3. Configuring the default source sets

You can further configure the default source sets to do things like add new directories, add filters, etc.

72.8.4. Adding extra source sets

If your Play application has additional sources that exist in non-standard directories, you can add extra source sets that Gradle will automatically add to the appropriate compile tasks.

Example 72.5. Adding extra source sets to a Play application

build.gradle

```
model {
    components {
        play {
            sources {
                java {
                    source.srcDir "additional/java"
                }
                javascript {
                    source {
                        srcDir "additional/javascript"
                        exclude "**/old_*.js"
                    }
                }
            }
        }
    }
}
```

build.gradle

```
model {
    components {
        play {
            sources {
                extraJava(JavaSourceSet) {
                    source.srcDir "extra/java"
                }
                extraTwirl(TwirlSourceSet) {
                    source.srcDir "extra/twirl"
                }
                extraRoutes(RoutesSourceSet) {
                    source.srcDir "extra/routes"
                }
            }
        }
    }
}
```

72.8.5. Configuring compiler options

If your Play application requires additional Scala compiler flags, you can add these arguments directly to the Scala compiler task.

Example 72.6. Configuring Scala compiler options

build.gradle

```
model {
    components {
        play {
            binaries.all {
                tasks.withType(PlatformScalaCompile) {
                    scalaCompileOptions.additionalParameters = ["-feature", "-language"]
                }
            }
        }
    }
}
```

72.8.6. Configuring routes style

The injected router is only supported in Play Framework 2.4 or better.

If your Play application's router uses dependency injection to access your controllers, you'll need to configure your application to not use the default static router. Under the covers, the Play plugin is using the `InjectedRoutesGenerator` instead of the default `StaticRoutesGenerator` to generate the router classes.

Example 72.7. Configuring routes style

build.gradle

```
model {
    components {
        play {
            injectedRoutesGenerator = true
        }
    }
}
```

72.8.7. Injecting a custom asset pipeline

Gradle Play support comes with a simplistic asset processing pipeline that minifies JavaScript assets. However, many organizations have their own custom pipeline for processing assets. You can easily hook the results of your pipeline into the Play binary by utilizing the `PublicAssets` property on the binary.

Example 72.8. Configuring a custom asset pipeline

build.gradle

```
model {
    components {
        play {
            binaries.all { binary ->
                tasks.create("addCopyrightToPlay${binary.name.capitalize()}Assets"
                    source "raw-assets"
                    copyrightFile = project.file('copyright.txt')
                    destinationDir = project.file("${buildDir}/play${binary.name.capitalize()}Assets")

                // Hook this task into the binary
                binary.assets.addAssetDir destinationDir
                binary.assets.builtBy copyrightTask
            }
        }
    }
}

class AddCopyrights extends SourceTask {
    @InputFile
    File copyrightFile

    @OutputDirectory
    File destinationDir

    @TaskAction
    void generateAssets() {
        String copyright = copyrightFile.text
        getSource().files.each { File file ->
            File outputFile = new File(destinationDir, file.name)
            outputFile.text = "${copyright}\n${file.text}"
        }
    }
}
```

72.9. Multi-project Play applications

Play applications can be built in multi-project builds as well. Simply apply the `play` plugin in the appropriate subprojects and create any project dependencies on the `play` configuration.

Example 72.9. Configuring dependencies on Play subprojects

build.gradle

```
dependencies {
    play project(":admin")
    play project(":user")
    play project(":util")
}
```

See the `play/multiproject` sample provided in the Gradle distribution for a working example.

72.10. Packaging a Play application for distribution

Gradle provides the capability to package your Play application so that it can easily be distributed and run in a target environment. The distribution package (zip file) contains the Play binary jars, all dependencies, and generated scripts that set up the classpath and run the application in a Play-specific Netty container.

The distribution can be created by running the `dist` lifecycle task and places the distribution in the `$buildDir` directory. Alternatively, one can validate the contents by running the `stage` lifecycle task which copies the files to the `$buildDir/stage` directory using the layout of the distribution package.

Table 72.5. Play distribution tasks

Task name	Depends on	Type
createPlayBinaryStartScripts	-	CreateStartScr
stagePlayBinaryDist	playBinary, createPlayBinaryStartScripts	Copy
createPlayBinaryZipDist		Zip
createPlayBinaryTarDist		Tar
stage	stagePlayBinaryDist	Task
dist	createPlayBinaryZipDist, createPlayBinaryTarDist	Task

72.10.1. Adding additional files to your Play application distribution

You can add additional files to the distribution package using the `Distribution API`.

Example 72.10. Add extra files to a Play application distribution

build.gradle

```
model {
    distributions {
        playBinary {
            contents {
                from("README.md")
                from("scripts") {
                    into "bin"
                }
            }
        }
    }
}
```

72.11. Building a Play application with an IDE

If you want to generate IDE metadata configuration for your Play project, you need to apply the appropriate IDE plugin. Gradle supports generating IDE metadata for IDEA only for Play projects at this time.

To generate IDEA's metadata, apply the `idea` plugin along with the `play` plugin.

Example 72.11. Applying both the Play and IDEA plugins

build.gradle

```
plugins {
    id 'play'
    id 'idea'
}
```

Source code generated by routes and Twirl templates cannot be generated by IDEA directly, so changes made to those files will not affect compilation until the next Gradle build. You can run the Play application with Gradle in continuous build to automatically rebuild and reload the application whenever something changes.

72.12. Resources

For additional information about developing Play applications:

- Play types in the Gradle DSL Guide:
 - `PlayApplicationBinarySpec`
 - `PlayApplicationSpec`
 - `PlayPlatform`
 - `JvmClasses`
 - `PublicAssets`
 - `PlayDistributionContainer`

- JavaScriptMinify
- PlayRun
- RoutesCompile
- TwirlCompile
- Play Framework Documentation.

Building native software

Support for building native software is currently incubating. Please be aware that the DSL, APIs and other configuration may change in later Gradle versions.

The native software plugins add support for building native software components, such as executables or shared libraries, from code written in C++, C and other languages. While many excellent build tools exist for this space of software development, Gradle offers developers its trademark power and flexibility together with dependency management practices more traditionally found in the JVM development space.

The native software plugins make use of the Gradle software model.

73.1. Features

The native software plugins provide:

- Support for building native libraries and applications on Windows, Linux, OS X and other platforms.
- Support for several source languages.
- Support for building different variants of the same software, for different architectures, operating systems, or for any purpose.
- Incremental parallel compilation, precompiled headers.
- Dependency management between native software components.
- Unit test execution.
- Generate Visual studio solution and project files.
- Deep integration with various tool chain, including discovery of installed tool chains.

73.2. Supported languages

The following source languages are currently supported:

- C
- C++
- Objective-C
- Objective-C++
- Assembly
- Windows resources

73.3. Tool chain support

Gradle offers the ability to execute the same build using different tool chains. When you build a native binary, Gradle will attempt to locate a tool chain installed on your machine that can build the binary. You can fine tune exactly how this works, see Section 73.18, “Tool chains” for details.

The following tool chains are supported:

Operating System	Tool Chain	Notes
Linux	GCC	
Linux	Clang	
Mac OS X	XCode	Uses the Clang tool chain bundled with XCode.
Windows	Visual C++	Windows XP and later, Visual C++ 2010/2012/2013/2015.
Windows	GCC with Cygwin 32	Windows XP and later.
Windows	GCC with MinGW	Windows XP and later. Mingw-w64 is currently not supported.

The following tool chains are unofficially supported. They generally work fine, but are not tested continuously:

Operating System	Tool Chain	Notes
Mac OS X	GCC from Macports	
Mac OS X	Clang from Macports	
Windows	GCC with Cygwin 64	Windows XP and later.
UNIX-like	GCC	
UNIX-like	Clang	

73.4. Tool chain installation

Note that if you are using GCC then you currently need to install support for C++, even if you are not building from C++ source. This restriction will be removed in a future Gradle version.

To build native software, you will need to have a compatible tool chain installed:

73.4.1. Windows

To build on Windows, install a compatible version of Visual Studio. The native plugins will discover the Visual Studio installations and select the latest version. There is no need to mess around with environment variables or batch scripts. This works fine from a Cygwin shell or the Windows command-line.

Alternatively, you can install Cygwin with GCC or MinGW. Clang is currently not supported.

73.4.2. OS X

To build on OS X, you should install XCode. The native plugins will discover the XCode installation using the system PATH.

The native plugins also work with GCC and Clang bundled with Macports. To use one of the Macports tool chains, you will need to make the tool chain the default using the `port select` command and add Macports to the system PATH.

73.4.3. Linux

To build on Linux, install a compatible version of GCC or Clang. The native plugins will discover GCC or Clang using the system PATH.

73.5. Native software model

The native software model builds on the base Gradle software model.

To build native software using Gradle, your project should define one or more *native components*. Each component represents either an executable or a library that Gradle should build. A project can define any number of components. Gradle does not define any components by default.

For each component, Gradle defines a *source set* for each language that the component can be built from. A source set is essentially just a set of source directories containing source files. For example, when you apply the `c` plugin and define a library called `helloworld`, Gradle will define, by default, a source set containing the C source files in the `src/helloworld/c` directory. It will use these source files to build the `helloworld` library. This is described in more detail below.

For each component, Gradle defines one or more *binaries* as output. To build a binary, Gradle will take the source files defined for the component, compile them as appropriate for the source language, and link the result into a binary file. For an executable component, Gradle can produce executable binary files. For a library component, Gradle can produce both static and shared library binary files. For example, when you define a library called `helloworld` and build on Linux, Gradle will, by default, produce `libhelloworld.sc` and `libhelloworld.a` binaries.

In many cases, more than one binary can be produced for a component. These binaries may vary based on the tool chain used to build, the compiler/linker flags supplied, the dependencies provided, or additional source files provided. Each native binary produced for a component is referred to as a *variant*. Binary variants are discussed in detail below.

73.6. Parallel Compilation

Gradle uses the single build worker pool to concurrently compile and link native components, by default. No special configuration is required to enable concurrent building.

By default, the worker pool size is determined by the number of available processors on the build machine (as reported to the build JVM). To explicitly set the number of workers use the `--max-workers` command-line option or `org.gradle.workers.max` system property. There is generally no need to change this setting from its default.

The build worker pool is shared across all build tasks. This means that when using parallel project execution, the maximum number of concurrent individual compilation operations does not increase. For example, if the build machine has 4 processing cores and 10 projects are compiling in parallel, Gradle will only use 4 total workers, not 40.

73.7. Building a library

To build either a static or shared native library, you define a library component in the `components` container. The following sample defines a library called `hello`:

Example 73.1. Defining a library component

build.gradle

```
model {
    components {
        hello(NativeLibrarySpec)
    }
}
```

A library component is represented using `NativeLibrarySpec`. Each library component can produce at least one shared library binary (`SharedLibraryBinarySpec`) and at least one static library binary (`StaticLibraryBinarySpec`).

73.8. Building an executable

To build a native executable, you define an executable component in the `components` container. The following sample defines an executable called `main`:

Example 73.2. Defining executable components

build.gradle

```
model {
    components {
        main(NativeExecutableSpec) {
            sources {
                c.lib library: "hello"
            }
        }
    }
}
```

An executable component is represented using `NativeExecutableSpec`. Each executable component can produce at least one executable binary (`NativeExecutableBinarySpec`).

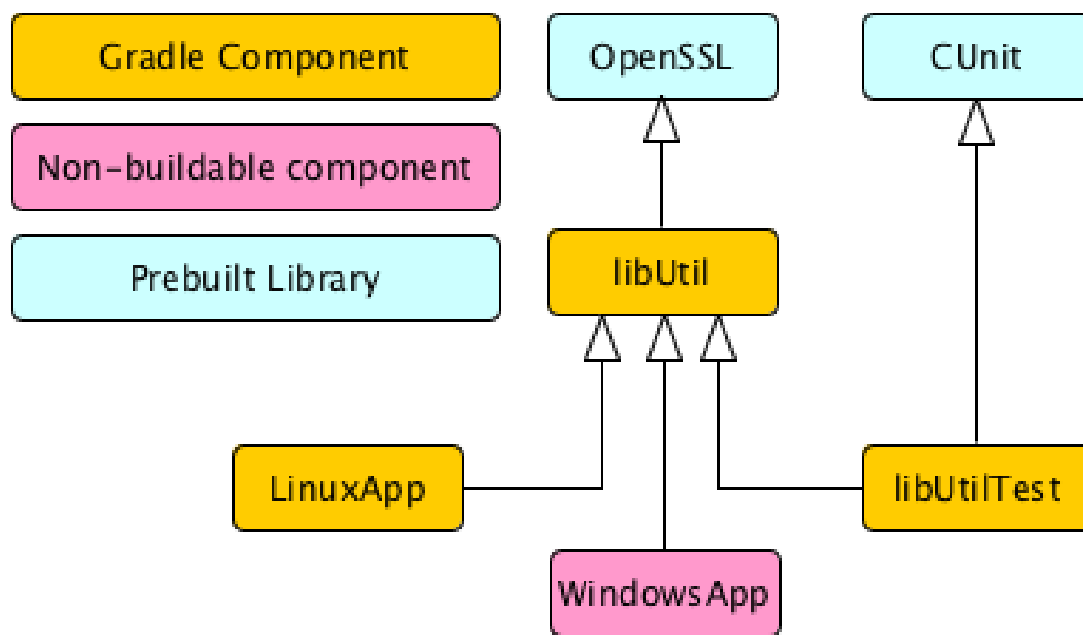
For each component defined, Gradle adds a `FunctionalSourceSet` with the same name. Each of these functional source sets will contain a language-specific source set for each of the languages supported by the project.

73.9. Assembling or building dependents

Sometimes, you may need to *assemble* (compile and link) or *build* (compile, link and test) a component or binary and its *dependents* (things that depend upon the component or binary). The native software model provides tasks that enable this capability. First, the *dependent components* report gives insight about the relationships between each component. Second, the *build and assemble dependents* tasks allow you to assemble or build a component and its dependents in one step.

In the following example, the build file defines `OpenSSL` as a dependency of `libUtil` and `libUtil` as a dependency of `LinuxApp` and `WindowsApp`. Test suites are treated similarly. Dependents can be thought of as reverse dependencies.

Figure 73.1. Dependent Components Example



By following the dependencies backwards, you can see `LinuxApp` and `WindowsApp` are *dependents* of `libUtil`. When `libUtil` is changed, Gradle will need to recompile or relink `LinuxApp` and `WindowsApp`.

When you *assemble* dependents of a component, the component and all of its dependents are compiled and linked, including any test suite binaries. Gradle's up-to-date checks are used to only compile or link if something has changed. For instance, if you have changed source files in a way that do not affect the headers of your project, Gradle will be able to skip compilation for dependent components and only need to re-link with the new library. Tests are not run when assembling a component.

When you *build* dependents of a component, the component and all of its dependent binaries are compiled, linked *and checked*. Checking components means running any check task including executing any test suites, so tests *are* run when building a component.

In the following sections, we will demonstrate the usage of the **`assembleDependents*`**, **`buildDependent*`** and **`dependentComponents`** tasks with a sample build that contains a CUnit test suite. The build script for the sample is the following:

Example 73.3. Sample build

build.gradle

```
apply plugin: "c"
apply plugin: 'cunit-test-suite'

model {
    flavors {
        passing
        failing
    }
    platforms {
        x86 {
            architecture "x86"
        }
    }
    components {
        operators(NativeLibrarySpec) {
            targetPlatform "x86"
        }
    }
    testSuites {
        operatorsTest(CUnitTestSuiteSpec) {
            testing $.components.operators
        }
    }
}
```

Note: The code for this example can be found at `samples/native-binaries/cunit` in the ‘-all’ distribution of Gradle.

73.9.1. Dependent components report

Gradle provides a report that you can run from the command-line that shows a graph of components in your project and components that depend upon them. The following is an example of running `gradle dependentComponents` on the sample project:

Example 73.4. Dependent components report

Output of **gradle dependentComponents**

```
> gradle dependentComponents
:dependentComponents

-----
Root project
-----

operators - Components that depend on native library 'operators'
+--- operators:failingSharedLibrary
+--- operators:failingStaticLibrary
+--- operators:passingSharedLibrary
\--- operators:passingStaticLibrary

Some test suites were not shown, use --test-suites or --all to show them.

BUILD SUCCESSFUL

Total time: 1 secs
```

See `DependentComponentsReport` API documentation for more details.

By default, non-buildable binaries and test suites are hidden from the report. The **dependentComponents** task provides options that allow you to see all dependents by using the `--all` option:

Example 73.5. Dependent components report

Output of **gradle dependentComponents --all**

```
> gradle dependentComponents --all
:dependentComponents

-----
Root project
-----

operators - Components that depend on native library 'operators'
+--- operators:failingSharedLibrary
+--- operators:failingStaticLibrary
|    \--- operatorsTest:failingCUnitExe (t)
+--- operators:passingSharedLibrary
\--- operators:passingStaticLibrary
     \--- operatorsTest:passingCUnitExe (t)

operatorsTest - Components that depend on Cunit test suite 'operatorsTest'
+--- operatorsTest:failingCUnitExe (t)
\--- operatorsTest:passingCUnitExe (t)

(t) - Test suite binary

BUILD SUCCESSFUL

Total time: 1 secs
```

Here is the corresponding report for the `operators` component, showing dependents of all its binaries:

Example 73.6. Report of components that depends on the operators component

Output of **gradle dependentComponents --component operators**

```
> gradle dependentComponents --component operators
:dependentComponents

-----
Root project
-----

operators - Components that depend on native library 'operators'
+--- operators:failingSharedLibrary
+--- operators:failingStaticLibrary
+--- operators:passingSharedLibrary
\--- operators:passingStaticLibrary

Some test suites were not shown, use --test-suites or --all to show them.

BUILD SUCCESSFUL

Total time: 1 secs
```

Here is the corresponding report for the `operators` component, showing dependents of all its binaries, including test suites:

Example 73.7. Report of components that depends on the operators component, including test suites

Output of **gradle dependentComponents --test-suites --component operators**

```
> gradle dependentComponents --test-suites --component operators
:dependentComponents

-----
Root project
-----

operators - Components that depend on native library 'operators'
+--- operators:failingSharedLibrary
+--- operators:failingStaticLibrary
|    \--- operatorsTest:failingCUnitExe (t)
+--- operators:passingSharedLibrary
\--- operators:passingStaticLibrary
     \--- operatorsTest:passingCUnitExe (t)

(t) - Test suite binary

BUILD SUCCESSFUL

Total time: 1 secs
```

73.9.2. Assembling dependents

For each `NativeBinarySpec`, Gradle will create a task named **`assembleDependents${component.name}`** that *assembles* (compile and link) the binary and all of its dependent binaries.

For each `NativeComponentSpec`, Gradle will create a task named **`assembleDependents${component}`** that assembles all the binaries of the component and all of their dependent binaries.

For example, to assemble the dependents of the "passing" flavor of the "static" library binary of the "operators" component, you would run the **`assembleDependentsOperatorsPassingStaticLibrary`** task:

Example 73.8. Assemble components that depends on the passing/static binary of the operators component

Output of **`gradle assembleDependentsOperatorsPassingStaticLibrary`**

```
> gradle assembleDependentsOperatorsPassingStaticLibrary
:compileOperatorsTestPassingCUnitExeOperatorsC
:operatorsTestCUnitLauncher
:compileOperatorsTestPassingCUnitExeOperatorsTestC
:compileOperatorsTestPassingCUnitExeOperatorsTestCUnitLauncher
:linkOperatorsTestPassingCUnitExe
:operatorsTestPassingCUnitExe
:assembleDependentsOperatorsTestPassingCUnitExe
:compileOperatorsPassingStaticLibraryOperatorsC
:createOperatorsPassingStaticLibrary
:operatorsPassingStaticLibrary
:assembleDependentsOperatorsPassingStaticLibrary

BUILD SUCCESSFUL

Total time: 1 secs
```

In the output above, the targeted binary gets assembled as well as the test suite binary that depends on it.

You can also assemble all of the dependents of a component (i.e. of all its binaries/variants) using the corresponding component task, e.g. **`assembleDependentsOperators`**. This is useful if you have many combinations of build types, flavors and platforms and want to assemble all of them.

73.9.3. Building dependents

For each `NativeBinarySpec`, Gradle will create a task named **`buildDependents${component.name}`** that builds (compile, link and check) the binary and all of its dependent binaries.

For each `NativeComponentSpec`, Gradle will create a task named **`buildDependents${component.name}`** that builds all the binaries of the component and all of their dependent binaries.

For example, to build the dependents of the "passing" flavor of the "static" library binary of the "operators" component, you would run the **`buildDependentsOperatorsPassingStaticLibrary`** task:

Example 73.9. Build components that depends on the passing/static binary of the operators component

Output of **gradle buildDependentsOperatorsPassingStaticLibrary**

```
> gradle buildDependentsOperatorsPassingStaticLibrary
:compileOperatorsTestPassingCUnitExeOperatorsC
:operatorsTestCUnitLauncher
:compileOperatorsTestPassingCUnitExeOperatorsTestC
:compileOperatorsTestPassingCUnitExeOperatorsTestCUnitLauncher
:linkOperatorsTestPassingCUnitExe
:operatorsTestPassingCUnitExe
:installOperatorsTestPassingCUnitExe
:runOperatorsTestPassingCUnitExe
:checkOperatorsTestPassingCUnitExe
:buildDependentsOperatorsTestPassingCUnitExe
:compileOperatorsPassingStaticLibraryOperatorsC
:createOperatorsPassingStaticLibrary
:operatorsPassingStaticLibrary
:buildDependentsOperatorsPassingStaticLibrary

BUILD SUCCESSFUL

Total time: 1 secs
```

In the output above, the targeted binary as well as the test suite binary that depends on it are built and the test suite has run.

You can also build all of the dependents of a component (i.e. of all its binaries/variants) using the corresponding component task, e.g. **buildDependentsOperators**.

73.10. Tasks

For each `NativeBinarySpec` that can be produced by a build, a single *lifecycle task* is constructed that can be used to create that binary, together with a set of other tasks that do the actual work of compiling, linking or assembling the binary.

Component Type	Native Binary Type	Lifecycle task	Location
<code>NativeExecutableSpec</code>	<code>NativeExecutableBinarySpec</code>	<code>\${component.name}Executable</code>	<code>\$(top)bin</code>
<code>NativeLibrarySpec</code>	<code>SharedLibraryBinarySpec</code>	<code>\${component.name}SharedLibrary</code>	<code>\$(top)lib</code>
<code>NativeLibrarySpec</code>	<code>StaticLibraryBinarySpec</code>	<code>\${component.name}StaticLibrary</code>	<code>\$(top)lib</code>

73.10.1. Check tasks

For each `NativeBinarySpec` that can be produced by a build, a single *check task* is constructed that can be used to assemble and check that binary.

Component Type	Native Binary Type	Check task
NativeExecutableSpec	NativeExecutableBinarySpec	check\${component.name}Ex
NativeLibrarySpec	SharedLibraryBinarySpec	check\${component.name}Sh
NativeLibrarySpec	StaticLibraryBinarySpec	check\${component.name}St

The built-in check task depends on all the *check tasks* for binaries in the project. Without either CUnit or Google plugins, the binary check task only depends on the *lifecycle task* that assembles the binary, see Section 73.10, “Tasks”.

When the CUnit or GoogleTest plugins are applied, the task that executes the test suites for a component are automatically wired to the appropriate *check task*.

You can also add custom check tasks as follows:

Example 73.10. Adding a custom check task

build.gradle

```

apply plugin: "cpp"
// You don't need to apply the plugin below if you're already using CUnit or GoogleI
apply plugin: TestingModelBasePlugin

task myCustomCheck {
    doLast {
        println 'Executing my custom check'
    }
}

model {
    components {
        hello(NativeLibrarySpec) {
            binaries.all {
                // Register our custom check task to all binaries of this component
                checkedBy $.tasks.myCustomCheck
            }
        }
    }
}

```

Note: The code for this example can be found at `samples/native-binaries/custom-check` in the ‘-all’ distribution of Gradle.

Now, running check or any of the *check tasks* for the hello binaries will run the custom check task:

Example 73.11. Running checks for a given binary

Output of **gradle checkHelloSharedLibrary**

```
> gradle checkHelloSharedLibrary
:myCustomCheck
Executing my custom check
:checkHelloSharedLibrary

BUILD SUCCESSFUL

Total time: 1 secs
```

73.10.2. Working with shared libraries

For each executable binary produced, the `cpp` plugin provides an `install${binary.name}` task, which creates a development install of the executable, along with the shared libraries it requires. This allows you to run the executable without needing to install the shared libraries in their final locations.

73.11. Finding out more about your project

Gradle provides a report that you can run from the command-line that shows some details about the components and binaries that your project produces. To use this report, just run **gradle components**. Below is an example of running this report for one of the sample projects:

Example 73.12. The components report

Output of **gradle components**

```
> gradle components
:components

-----
Root project
-----

Native library 'hello'
-----

Source sets
  C++ source 'hello:cpp'
    srcDir: src/hello/cpp

Binaries
  Shared library 'hello:sharedLibrary'
    build using task: :helloSharedLibrary
    build type: build type 'debug'
    flavor: flavor 'default'
    target platform: platform 'current'
    tool chain: Tool chain 'clang' (Clang)
    shared library file: build/libs/hello/shared/libhello.dylib
  Static library 'hello:staticLibrary'
    build using task: :helloStaticLibrary
    build type: build type 'debug'
    flavor: flavor 'default'
    target platform: platform 'current'
    tool chain: Tool chain 'clang' (Clang)
    static library file: build/libs/hello/static/libhello.a

Native executable 'main'
-----

Source sets
  C++ source 'main:cpp'
    srcDir: src/main/cpp

Binaries
  Executable 'main:executable'
    build using task: :mainExecutable
    install using task: :installMainExecutable
    build type: build type 'debug'
    flavor: flavor 'default'
    target platform: platform 'current'
    tool chain: Tool chain 'clang' (Clang)
    executable file: build/exe/main/main

Note: currently not all plugins register their components, so some components may nc

BUILD SUCCESSFUL

Total time: 1 secs
```

73.12. Language support

Presently, Gradle supports building native software from any combination of source languages listed below. A native binary project will contain one or more named `FunctionalSourceSet` instances (eg 'main', 'test', etc), each of which can contain `LanguageSourceSets` containing source files, one for each language.

- C
- C++
- Objective-C
- Objective-C++
- Assembly
- Windows resources

73.12.1. C++ sources

C++ language support is provided by means of the 'cpp' plugin.

Example 73.13. The 'cpp' plugin

build.gradle

```
apply plugin: 'cpp'
```

C++ sources to be included in a native binary are provided via a `CppSourceSet`, which defines a set of C++ source files and optionally a set of exported header files (for a library). By default, for any named component the `CppSourceSet` contains .cpp source files in `src/${name}/cpp`, and header files in `src/${name}/headers`.

While the `cpp` plugin defines these default locations for each `CppSourceSet`, it is possible to extend or override these defaults to allow for a different project layout.

Example 73.14. C++ source set

build.gradle

```
sources {  
    cpp {  
        source {  
            srcDir "src/source"  
            include "**/*.cpp"  
        }  
    }  
}
```

For a library named 'main', header files in `src/main/headers` are considered the “public” or “exported” headers. Header files that should not be exported should be placed inside the `src/main/cpp` directory (though be aware that such header files should always be referenced in a manner relative to the file including them).

73.12.2. C sources

C language support is provided by means of the 'c' plugin.

Example 73.15. The 'c' plugin

build.gradle

```
apply plugin: 'c'
```

C sources to be included in a native binary are provided via a `CSourceSet`, which defines a set of C source files and optionally a set of exported header files (for a library). By default, for any named component the `CSourceSet` contains `.c` source files in `src/${name}/c`, and header files in `src/${name}`.

While the `c` plugin defines these default locations for each `CSourceSet`, it is possible to extend or override these defaults to allow for a different project layout.

Example 73.16. C source set

build.gradle

```
sources {  
    c {  
        source {  
            srcDir "src/source"  
            include "**/*.c"  
        }  
        exportedHeaders {  
            srcDir "src/include"  
        }  
    }  
}
```

For a library named 'main', header files in `src/main/headers` are considered the “public” or “exported” headers. Header files that should not be exported should be placed inside the `src/main/c` directory (though be aware that such header files should always be referenced in a manner relative to the file including them).

73.12.3. Assembler sources

Assembly language support is provided by means of the 'assembler' plugin.

Example 73.17. The 'assembler' plugin

build.gradle

```
apply plugin: 'assembler'
```

Assembler sources to be included in a native binary are provided via a `AssemblerSourceSet`, which defines a set of Assembler source files. By default, for any named component the `AssemblerSourceSet` contains `.s` source files under `src/${name}/asm`.

73.12.4. Objective-C sources

Objective-C language support is provided by means of the 'objective-c' plugin.

Example 73.18. The 'objective-c' plugin

build.gradle

```
apply plugin: 'objective-c'
```

Objective-C sources to be included in a native binary are provided via a `ObjectiveCSourceSet`, which defines a set of Objective-C source files. By default, for any named component the `ObjectiveCSourceSet` contains `.m` source files under `src/${name}/objectiveC`.

73.12.5. Objective-C++ sources

Objective-C++ language support is provided by means of the 'objective-cpp' plugin.

Example 73.19. The 'objective-cpp' plugin

build.gradle

```
apply plugin: 'objective-cpp'
```

Objective-C++ sources to be included in a native binary are provided via a `ObjectiveCppSourceSet`, which defines a set of Objective-C++ source files. By default, for any named component the `ObjectiveCppSourceSet` contains `.mm` source files under `src/${name}/objectiveCpp`.

73.13. Configuring the compiler, assembler and linker

Each binary to be produced is associated with a set of compiler and linker settings, which include command-line arguments as well as macro definitions. These settings can be applied to all binaries, an individual binary, or selectively to a group of binaries based on some criteria.

Example 73.20. Settings that apply to all binaries

build.gradle

```
model {
    binaries {
        all {
            // Define a preprocessor macro for every binary
            cppCompiler.define "NDEBUG"

            // Define toolchain-specific compiler and linker options
            if (toolChain in Gcc) {
                cppCompiler.args "-O2", "-fno-access-control"
                linker.args "-Xlinker", "-S"
            }
            if (toolChain in VisualCpp) {
                cppCompiler.args "/Zi"
                linker.args "/DEBUG"
            }
        }
    }
}
```

Each binary is associated with a particular NativeToolChain, allowing settings to be targeted based on this value.

It is easy to apply settings to all binaries of a particular type:

Example 73.21. Settings that apply to all shared libraries

build.gradle

```
// For any shared library binaries built with Visual C++,
// define the DLL_EXPORT macro
model {
    binaries {
        withType(SharedLibraryBinarySpec) {
            if (toolChain in VisualCpp) {
                cCompiler.args "/Zi"
                cCompiler.define "DLL_EXPORT"
            }
        }
    }
}
```

Furthermore, it is possible to specify settings that apply to all binaries produced for a particular executable or library component:

Example 73.22. Settings that apply to all binaries produced for the 'main' executable component

build.gradle

```
model {
    components {
        main(NativeExecutableSpec) {
            targetPlatform "x86"
            binaries.all {
                if (toolChain in VisualCpp) {
                    sources {
                        platformAsm(AssemblerSourceSet) {
                            source.srcDir "src/main/asm_i386_masm"
                        }
                    }
                    assembler.args "/Zi"
                } else {
                    sources {
                        platformAsm(AssemblerSourceSet) {
                            source.srcDir "src/main/asm_i386_gcc"
                        }
                    }
                    assembler.args "-g"
                }
            }
        }
    }
}
```

The example above will apply the supplied configuration to all executable binaries built.

Similarly, settings can be specified to target binaries for a component that are of a particular type: eg all shared libraries for the main library component.

Example 73.23. Settings that apply only to shared libraries produced for the 'main' library component

build.gradle

```
model {
    components {
        main(NativeLibrarySpec) {
            binaries.withType(SharedLibraryBinarySpec) {
                // Define a preprocessor macro that only applies to shared libraries
                cppCompiler.define "DLL_EXPORT"
            }
        }
    }
}
```

73.14. Windows Resources

When using the VisualCpp tool chain, Gradle is able to compile Window Resource (rc) files and link them into a native binary. This functionality is provided by the 'windows-resources' plugin.

Example 73.24. The 'windows-resources' plugin

build.gradle

```
apply plugin: 'windows-resources'
```

Windows resources to be included in a native binary are provided via a `WindowsResourceSet`, which defines a set of Windows Resource source files. By default, for any named component the `WindowsResourceSet` contains `.rc` source files under `src/${name}/rc`.

As with other source types, you can configure the location of the windows resources that should be included in the binary.

Example 73.25. Configuring the location of Windows resource sources

build-resource-only-dll.gradle

```
sources {
    rc {
        source {
            srcDirs "src/hello/rc"
        }
        exportedHeaders {
            srcDirs "src/hello/headers"
        }
    }
}
```

You are able to construct a resource-only library by providing Windows Resource sources with no other language sources, and configure the linker as appropriate:

Example 73.26. Building a resource-only dll

build-resource-only-dll.gradle

```
model {
    components {
        helloRes(NativeLibrarySpec) {
            binaries.all {
                rcCompiler.args "/v"
                linker.args "/noentry", "/machine:x86"
            }
            sources {
                rc {
                    source {
                        srcDirs "src/hello/rc"
                    }
                    exportedHeaders {
                        srcDirs "src/hello/headers"
                    }
                }
            }
        }
    }
}
```

The example above also demonstrates the mechanism of passing extra command-line arguments to the resource compiler. The `rcCompiler` extension is of type `PreprocessingTool`.

73.15. Library Dependencies

Dependencies for native components are binary libraries that export header files. The header files are used during compilation, with the compiled binary dependency being used during linking and execution. Header files should be organized into subdirectories to prevent clashes of commonly named headers. For instance, if your `mylib` project has a `logging.h` header, it will make it less likely the wrong header is used if you include it as `"mylib/logging.h"` instead of `"logging.h"`.

73.15.1. Dependencies within the same project

A set of sources may depend on header files provided by another binary component within the same project. A common example is a native executable component that uses functions provided by a separate native library component.

Such a library dependency can be added to a source set associated with the `executable` component:

Example 73.27. Providing a library dependency to the source set

build.gradle

```
sources {  
    cpp {  
        lib library: "hello"  
    }  
}
```

Alternatively, a library dependency can be provided directly to the `NativeExecutableBinarySpec` for the `executable`.

Example 73.28. Providing a library dependency to the binary

build.gradle

```
model {
    components {
        hello(NativeLibrarySpec) {
            sources {
                c {
                    source {
                        srcDir "src/source"
                        include "**/*.c"
                    }
                    exportedHeaders {
                        srcDir "src/include"
                    }
                }
            }
        }
        main(NativeExecutableSpec) {
            sources {
                cpp {
                    source {
                        srcDir "src/source"
                        include "**/*.cpp"
                    }
                }
            }
            binaries.all {
                // Each executable binary produced uses the 'hello' static library k
                lib library: 'hello', linkage: 'static'
            }
        }
    }
}
```

73.15.2. Project Dependencies

For a component produced in a different Gradle project, the notation is similar.

Example 73.29. Declaring project dependencies

build.gradle

```
project(":lib") {
    apply plugin: "cpp"
    model {
        components {
            main(NativeLibrarySpec)
        }

        // For any shared library binaries built with Visual C++,
        // define the DLL_EXPORT macro
        binaries {
            withType(SharedLibraryBinarySpec) {
                if (toolChain in VisualCpp) {
                    cppCompiler.define "DLL_EXPORT"
                }
            }
        }
    }
}

project(":exe") {
    apply plugin: "cpp"

    model {
        components {
            main(NativeExecutableSpec) {
                sources {
                    cpp {
                        lib project: ':lib', library: 'main'
                    }
                }
            }
        }
    }
}
```

73.16. Precompiled Headers

Precompiled headers are a performance optimization that reduces the cost of compiling widely used headers multiple times. This feature *precompiles* a header such that the compiled object file can be reused when compiling each source file rather than recompiling the header each time. This support is available for C, C++, Objective-C, and Objective-C++ builds.

To configure a precompiled header, first a header file needs to be defined that includes all of the headers that should be precompiled. It must be specified as the first included header in every source file where the precompiled header should be used. It is assumed that this header file, and any headers it contains, make use of header guards so that they can be included in an idempotent manner. If header guards are not used in a header file, it is possible the header could be compiled more than once and could potentially lead to a broken build.

Example 73.30. Creating a precompiled header file

src/hello/headers/pch.h

```
#ifndef PCH_H
#define PCH_H
#include <iostream>
#include "hello.h"
#endif
```

Example 73.31. Including a precompiled header file in a source file

src/hello/cpp/hello.cpp

```
#include "pch.h"

void LIB_FUNC Greeter::hello () {
    std::cout << "Hello world!" << std::endl;
}
```

Precompiled headers are specified on a source set. Only one precompiled header file can be specified on a given source set and will be applied to all source files that declare it as the first include. If a source file does not include this header file as the first header, the file will be compiled in the normal manner (without making use of the precompiled header object file). The string provided should be the same as that which is used in the "#include" directive in the source files.

Example 73.32. Configuring a precompiled header

build.gradle

```
model {
    components {
        hello(NativeLibrarySpec) {
            sources {
                cpp {
                    preCompiledHeader "pch.h"
                }
            }
        }
    }
}
```

A precompiled header must be included in the same way for all files that use it. Usually, this means the header file should exist in the source set "headers" directory or in a directory included on the compiler include path.

73.17. Native Binary Variants

For each executable or library defined, Gradle is able to build a number of different native binary variants. Examples of different variants include debug vs release binaries, 32-bit vs 64-bit binaries, and binaries produced with different custom preprocessor flags.

Binaries produced by Gradle can be differentiated on build type, platform, and flavor. For each of these

'variant dimensions', it is possible to specify a set of available values as well as target each component at one, some or all of these. For example, a plugin may define a range of support platforms, but you may choose to only target Windows-x86 for a particular component.

73.17.1. Build types

A `build type` determines various non-functional aspects of a binary, such as whether debug information is included, or what optimisation level the binary is compiled with. Typical build types are 'debug' and 'release', but a project is free to define any set of build types.

Example 73.33. Defining build types

build.gradle

```
model {
    buildTypes {
        debug
        release
    }
}
```

If no build types are defined in a project, then a single, default build type called 'debug' is added.

For a build type, a Gradle project will typically define a set of compiler/linker flags per tool chain.

Example 73.34. Configuring debug binaries

build.gradle

```
model {
    binaries {
        all {
            if (toolChain in Gcc && buildType == buildTypes.debug) {
                cppCompiler.args "-g"
            }
            if (toolChain in VisualCpp && buildType == buildTypes.debug) {
                cppCompiler.args '/Zi'
                cppCompiler.define 'DEBUG'
                linker.args '/DEBUG'
            }
        }
    }
}
```

At this stage, it is completely up to the build script to configure the relevant compiler/linker flags for each build type. Future versions of Gradle will automatically include the appropriate debug flags for any 'debug' build type, and may be aware of various levels of optimisation as well.

73.17.2. Platform

An executable or library can be built to run on different operating systems and cpu architectures, with a variant being produced for each platform. Gradle defines each OS/architecture combination as a `NativePlatform`, and a project may define any number of platforms. If no platforms are defined in a project, then a single, default platform 'current' is added.

Presently, a `Platform` consists of a defined operating system and architecture. As we continue to develop the native binary support in Gradle, the concept of `Platform` will be extended to include things like C-runtime version, Windows SDK, ABI, etc. Sophisticated builds may use the extensibility of Gradle to apply additional attributes to each platform, which can then be queried to specify particular includes, preprocessor macros or compiler arguments for a native binary.

Example 73.35. Defining platforms

build.gradle

```
model {
    platforms {
        x86 {
            architecture "x86"
        }
        x64 {
            architecture "x86_64"
        }
        itanium {
            architecture "ia-64"
        }
    }
}
```

For a given variant, Gradle will attempt to find a `NativeToolChain` that is able to build for the target platform. Available tool chains are searched in the order defined. See the tool chains section below for more details.

73.17.3. Flavor

Each component can have a set of named `flavors`, and a separate binary variant can be produced for each flavor. While the `build type` and `target platform` variant dimensions have a defined meaning in Gradle, each project is free to define any number of flavors and apply meaning to them in any way.

An example of component flavors might differentiate between 'demo', 'paid' and 'enterprise' editions of the component, where the same set of sources is used to produce binaries with different functions.

Example 73.36. Defining flavors

build.gradle

```
model {
    flavors {
        english
        french
    }
    components {
        hello(NativeLibrarySpec) {
            binaries.all {
                if (flavor == flavors.french) {
                    cppCompiler.define "FRENCH"
                }
            }
        }
    }
}
```

In the example above, a library is defined with a 'english' and 'french' flavor. When compiling the 'french' variant, a separate macro is defined which leads to a different binary being produced.

If no flavor is defined for a component, then a single default flavor named 'default' is used.

73.17.4. Selecting the build types, platforms and flavors for a component

For a default component, Gradle will attempt to create a native binary variant for each and every combination of buildType and flavor defined for the project. It is possible to override this on a per-component basis, by specifying the set of targetBuildTypes and/or targetFlavors. By default, Gradle will build for the default platform, see above, unless specified explicitly on a per-component basis by specifying a set of targetPlatforms.

Example 73.37. Targeting a component at particular platforms

build.gradle

```
model {
    components {
        hello(NativeLibrarySpec) {
            targetPlatform "x86"
            targetPlatform "x64"
        }
        main(NativeExecutableSpec) {
            targetPlatform "x86"
            targetPlatform "x64"
            sources {
                cpp.lib library: 'hello', linkage: 'static'
            }
        }
    }
}
```

Here you can see that the TargetedNativeComponent.targetPlatform(java.lang.String) method is used to

specify a platform that the `NativeExecutableSpec` named `main` should be built for.

A similar mechanism exists for selecting `TargetedNativeComponent.targetBuildTypes(java.lang.String[])` and `TargetedNativeComponent.targetFlavors(java.lang.String[])`.

73.17.5. Building all possible variants

When a set of build types, target platforms, and flavors is defined for a component, a `NativeBinarySpec` model element is created for every possible combination of these. However, in many cases it is not possible to build a particular variant, perhaps because no tool chain is available to build for a particular platform.

If a binary variant cannot be built for any reason, then the `NativeBinarySpec` associated with that variant will not be buildable. It is possible to use this property to create a task to generate all possible variants on a particular machine.

Example 73.38. Building all possible variants

build.gradle

```
model {
    tasks {
        buildAllExecutables(Task) {
            dependsOn $.binaries.findAll { it.buildable }
        }
    }
}
```

73.18. Tool chains

A single build may utilize different tool chains to build variants for different platforms. To this end, the core 'native-binary' plugins will attempt to locate and make available supported tool chains. However, the set of tool chains for a project may also be explicitly defined, allowing additional cross-compilers to be configured as well as allowing the install directories to be specified.

73.18.1. Defining tool chains

The supported tool chain types are:

- Gcc
- Clang
- VisualCpp

Example 73.39. Defining tool chains

build.gradle

```
model {
    toolChains {
        visualCpp(VisualCpp) {
            // Specify the installDir if Visual Studio cannot be located
            // installDir "C:/Apps/Microsoft Visual Studio 10.0"
        }
        gcc(Gcc) {
            // Uncomment to use a GCC install that is not in the PATH
            // path "/usr/bin/gcc"
        }
        clang(Clang)
    }
}
```

Each tool chain implementation allows for a certain degree of configuration (see the API documentation for more details).

73.18.2. Using tool chains

It is not necessary or possible to specify the tool chain that should be used to build. For a given variant, Gradle will attempt to locate a `NativeToolChain` that is able to build for the target platform. Available tool chains are searched in the order defined.

When a platform does not define an architecture or operating system, the default target of the tool chain is assumed. So if a platform does not define a value for `operatingSystem`, Gradle will find the first available tool chain that can build for the specified architecture.

The core Gradle tool chains are able to target the following architectures out of the box. In each case, the tool chain will target the current operating system. See the next section for information on cross-compiling for other operating systems.

Tool Chain	Architectures
GCC	x86, x86_64
Clang	x86, x86_64
Visual C++	x86, x86_64, ia-64

So for GCC running on linux, the supported target platforms are 'linux/x86' and 'linux/x86_64'. For GCC running on Windows via Cygwin, platforms 'windows/x86' and 'windows/x86_64' are supported. (The Cygwin POSIX runtime is not yet modelled as part of the platform, but will be in the future.)

If no target platforms are defined for a project, then all binaries are built to target a default platform named 'current'. This default platform does not specify any `architecture` or `operatingSystem` value, hence using the default values of the first available tool chain.

Gradle provides a [*hook*](#) that allows the build author to control the exact set of arguments passed to a tool

chain executable. This enables the build author to work around any limitations in Gradle, or assumptions that Gradle makes. The arguments hook should be seen as a 'last-resort' mechanism, with preference given to truly modelling the underlying domain.

Example 73.40. Reconfigure tool arguments

build.gradle

```
model {
    toolChains {
        visualCpp(VisualCpp) {
            eachPlatform {
                cppCompiler.withArguments { args ->
                    args << "-DFRENCH"
                }
            }
        }
        clang(Clang) {
            eachPlatform {
                cCompiler.withArguments { args ->
                    Collections.replaceAll(args, "CUSTOM", "-DFRENCH")
                }
                linker.withArguments { args ->
                    args.remove "CUSTOM"
                }
                staticLibArchiver.withArguments { args ->
                    args.remove "CUSTOM"
                }
            }
        }
    }
}
```

73.18.3. Cross-compiling with GCC

Cross-compiling is possible with the `Gcc` and `Clang` tool chains, by adding support for additional target platforms. This is done by specifying a target platform for a toolchain. For each target platform a custom configuration can be specified.

Example 73.41. Defining target platforms

build.gradle

```
model {
    toolChains {
        gcc(Gcc) {
            target("arm"){
                cppCompiler.withArguments { args ->
                    args << "-m32"
                }
                linker.withArguments { args ->
                    args << "-m32"
                }
            }
            target("sparc")
        }
    }
    platforms {
        arm {
            architecture "arm"
        }
        sparc {
            architecture "sparc"
        }
    }
    components {
        main(NativeExecutableSpec) {
            targetPlatform "arm"
            targetPlatform "sparc"
        }
    }
}
```

73.19. Visual Studio IDE integration

Gradle has the ability to generate Visual Studio project and solution files for the native components defined in your build. This ability is added by the `visual-studio` plugin. For a multi-project build, all projects with native components should have this plugin applied.

When the `visual-studio` plugin is applied, a task name `${component.name}VisualStudio` is created for each defined component. This task will generate a Visual Studio Solution file for the named component. This solution will include a Visual Studio Project for that component, as well as linking to project files for each depended-on binary.

The content of the generated visual studio files can be modified via API hooks, provided by the `visualStudio` extension. Take a look at the 'visual-studio' sample, or see `VisualStudioExtension.getProjects()` and `VisualStudioExtension.getSolutions()` in the API documentation for more details.

73.20. CUnit support

The Gradle `cunit` plugin provides support for compiling and executing CUnit tests in your native-binary project. For each `NativeExecutableSpec` and `NativeLibrarySpec` defined in your project, Gradle will create a matching `CUnitTestSuiteSpec` component, named `${component.name}Test`.

73.20.1. CUnit sources

Gradle will create a `CSourceSet` named 'cunit' for each `CUnitTestSuiteSpec` component in the project. This source set should contain the cunit test files for the component under test. Source files can be located in the conventional location (`src/${component.name}Test/cunit`) or can be configured like any other source set.

Gradle initialises the CUnit test registry and executes the tests, utilising some generated CUnit launcher sources. Gradle will expect and call a function with the signature `void gradle_cunit_register()` that you can use to configure the actual CUnit suites and tests to execute.

Due to this mechanism, your CUnit sources may not contain a `main` method since this will clash with the method provided by Gradle.

73.20.2. Building CUnit executables

A `CUnitTestSuiteSpec` component has an associated `NativeExecutableSpec` or `NativeLibrarySpec` component. For each `NativeBinarySpec` configured for the main component, a matching `CUnitTestSuiteBinarySpec` will be configured on the test suite component. These test suite binaries can be configured in a similar way to any other binary instance:

Example 73.42. Registering CUnit tests

suite_operators.c

```
#include <CUnit/Basic.h>
#include "gradle_cunit_register.h"
#include "test_operators.h"

int suite_init(void) {
    return 0;
}

int suite_clean(void) {
    return 0;
}

void gradle_cunit_register() {
    CU_pSuite pSuiteMath = CU_add_suite("operator tests", suite_init, suite_clean);
    CU_add_test(pSuiteMath, "test_plus", test_plus);
    CU_add_test(pSuiteMath, "test_minus", test_minus);
}
```

build.gradle

```
model {
    binaries {
        withType(CUnitTestSuiteBinarySpec) {
            lib library: "cunit", linkage: "static"

            if (flavor == flavors.failing) {
                cCompiler.define "PLUS_BROKEN"
            }
        }
    }
}
```

Both the CUnit sources provided by your project and the generated launcher require the core CUnit headers and libraries. Presently, this library dependency must be provided by your project for each CUnitTestSuiteBinarySpec.

73.20.3. Running CUnit tests

For each CUnitTestSuiteBinarySpec, Gradle will create a task to execute this binary, which will run all of the registered CUnit tests. Test results will be found in the `${build.dir}/test-results` directory.

Example 73.43. Running CUnit tests

build.gradle

```

apply plugin: "c"
apply plugin: 'cunit-test-suite'

model {
    flavors {
        passing
        failing
    }
    platforms {
        x86 {
            architecture "x86"
        }
    }
    repositories {
        libs(PrebuiltLibraries) {
            cunit {
                headers.srcDir "libs/cunit/2.1-2/include"
                binaries.withType(StaticLibraryBinary) {
                    staticLibraryFile =
                        file("libs/cunit/2.1-2/lib/" +
                            findCUnitLibForPlatform(targetPlatform))
                }
            }
        }
    }
    components {
        operators(NativeLibrarySpec) {
            targetPlatform "x86"
        }
    }
    testSuites {
        operatorsTest(CUnitTestSuiteSpec) {
            testing $.components.operators
        }
    }
}

model {
    binaries {
        withType(CUnitTestSuiteBinarySpec) {
            lib library: "cunit", linkage: "static"

            if (flavor == flavors.failing) {
                cCompiler.define "PLUS_BROKEN"
            }
        }
    }
}

```

Note: The code for this example can be found at `samples/native-binaries/cunit` in the ‘-all’ distribution of Gradle.

Output of `gradle -q runOperatorsTestFailingCUnitExe`


```
> gradle -q runOperatorsTestFailingCUnitExe
```

There were test failures:

1. /home/user/gradle/samples/native-binaries/cunit/src/operatorsTest/c/test_plus.c
2. /home/user/gradle/samples/native-binaries/cunit/src/operatorsTest/c/test_plus.c

The current support for CUnit is quite rudimentary. Plans for future integration include:

- Allow tests to be declared with Javadoc-style annotations.
- Improved HTML reporting, similar to that available for JUnit.
- Real-time feedback for test execution.
- Support for additional test frameworks.

73.21. GoogleTest support

The Gradle `google-test` plugin provides support for compiling and executing GoogleTest tests in your native-binary project. For each `NativeExecutableSpec` and `NativeLibrarySpec` defined in your project, Gradle will create a matching `GoogleTestTestSuiteSpec` component, named `${component}.nativeBinaryGoogleTestTestSuiteSpec`.

73.21.1. GoogleTest sources

Gradle will create a `CppSourceSet` named 'cpp' for each `GoogleTestTestSuiteSpec` component in the project. This source set should contain the GoogleTest test files for the component under test. Source files can be located in the conventional location (`src/${component.name}Test/cpp`) or can be configured like any other source set.

73.21.2. Building GoogleTest executables

A `GoogleTestTestSuiteSpec` component has an associated `NativeExecutableSpec` or `NativeLibrarySpec` component. For each `NativeBinarySpec` configured for the main component, a matching `GoogleTestTestSuiteBinarySpec` will be configured on the test suite component. These test suite binaries can be configured in a similar way to any other binary instance:

Example 73.44. Registering GoogleTest tests

build.gradle

```
model {
    binaries {
        withType(GoogleTestTestSuiteBinarySpec) {
            lib library: "googleTest", linkage: "static"

            if (flavor == flavors.failing) {
                cppCompiler.define "PLUS_BROKEN"
            }

            if (targetPlatform.operatingSystem.linux) {
                cppCompiler.args '-pthread'
                linker.args '-pthread'
            }
        }
    }
}
```

Note: The code for this example can be found at `samples/native-binaries/google-test` in the ‘-all’ distribution of Gradle.

The GoogleTest sources provided by your project require the core GoogleTest headers and libraries. Presently, this library dependency must be provided by your project for each `GoogleTestTestSuiteBinarySpec`.

73.21.3. Running GoogleTest tests

For each `GoogleTestTestSuiteBinarySpec`, Gradle will create a task to execute this binary, which will run all of the registered GoogleTest tests. Test results will be found in the `${build.dir}/test-result` directory.

The current support for GoogleTest is quite rudimentary. Plans for future integration include:

- Improved HTML reporting, similar to that available for JUnit.
- Real-time feedback for test execution.
- Support for additional test frameworks.

Extending the software model

Support for the software model is currently incubating. Please be aware that the DSL, APIs and other configuration may change in later Gradle versions.

One of the strengths of Gradle has always been its extensibility, and its adaptability to new domains. The software model takes this extensibility to a new level, enabling the deep modeling of specific domains via richly typed DSLs. The following chapter describes how the model and the corresponding DSLs can be extended to support domains like Java, Play Framework or native software development. Before reading this you should be familiar with the Gradle software model rule based configuration and concepts.

The following build script is an example of using a custom software model for building Markdown based documentation:

Example 74.1. an example of using a custom software model

build.gradle

```
import sample.documentation.DocumentationComponent
import sample.documentation.TextSourceSet
import sample.markdown.MarkdownSourceSet

apply plugin:sample.documentation.DocumentationPlugin
apply plugin:sample.markdown.MarkdownPlugin

model {
    components {
        docs(DocumentationComponent) {
            sources {
                reference(TextSourceSet)
                userguide(MarkdownSourceSet) {
                    generateIndex = true
                    smartQuotes = true
                }
            }
        }
    }
}
```

Note: The code for this example can be found at `samples/customModel/languageType/` in the ‘-all’ distribution of Gradle.

The rest of this chapter is dedicated to explaining what is going on behind this build script.

74.1. Concepts

A custom software model type has a public type, a base interface and internal views. Multiple such types then collaborate to define a custom software model.

74.1.1. Public type and base interfaces

Extended types declare a public type that extends a base interface:

- Components extend the `ComponentSpec` base interface
- Binaries extend the `BinarySpec` base interface
- Source sets extend the `LanguageSourceSet` base interface

The public type is exposed to build logic.

74.1.2. Internal views

Adding internal views to your model type, you can make some data visible to build logic via a public type, while hiding the rest of the data behind the internal view types. This is covered in a dedicated section below.

74.1.3. Components all the way down

Components are composed of other components. A source set is just a special kind of component representing sources. It might be that the sources are provided, or generated. Similarly, some components are composed of different binaries, which are built by tasks. All buildable components are built by tasks. In the software model, you will write rules to generate both binaries from components and tasks from binaries.

74.2. Components

To declare a custom component type one must extend `ComponentSpec`, or one of the following, depending on the use case:

- `SourceComponentSpec` represents a component which has sources
- `VariantComponentSpec` represents a component which generates different binaries based on context (target platforms, build flavors, ...). Such a component generally produces multiple binaries.
- `GeneralComponentSpec` is a convenient base interface for components that are built from sources and variant-aware. This is the typical case for a lot of software components, and therefore it should be in most of the cases the base type to be extended.

The core software model includes more types that can be used as base for extension. For example: `LibrarySpec` and `ApplicationSpec` can also be extended in this manner. These are no-op extensions of `GeneralComponentSpec` used to describe a software model better by distinguishing libraries and applications components. `TestSuiteSpec` should be used for all components that describe a test suite.

Example 74.2. Declare a custom component

DocumentationComponent.groovy

```
@Managed
interface DocumentationComponent extends GeneralComponentSpec {}
```

Types extending ComponentSpec are registered via a rule annotated with ComponentType:

Example 74.3. Register a custom component

DocumentationPlugin.groovy

```
class DocumentationPlugin extends RuleSource {
    @ComponentType
    void registerComponent(TypeBuilder<DocumentationComponent> builder) {}
}
```

74.3. Binaries

To declare a custom binary type one must extend BinarySpec.

Example 74.4. Declare a custom binary

DocumentationBinary.groovy

```
@Managed
interface DocumentationBinary extends BinarySpec {
    File getOutputDir()
    void setOutputDir(File outputDir)
}
```

Types extending BinarySpec are registered via a rule annotated with ComponentType:

Example 74.5. Register a custom binary

DocumentationPlugin.groovy

```
class DocumentationPlugin extends RuleSource {
    @ComponentType
    void registerBinary(TypeBuilder<DocumentationBinary> builder) {}
}
```

74.4. Source sets

To declare a custom source set type one must extend LanguageSourceSet.

Example 74.6. Declare a custom source set

MarkdownSourceSet.groovy

```
@Managed
interface MarkdownSourceSet extends LanguageSourceSet {
    boolean isGenerateIndex()
    void setGenerateIndex(boolean generateIndex)

    boolean isSmartQuotes()
    void setSmartQuotes(boolean smartQuotes)
}
```

Types extending `LanguageSourceSet` are registered via a rule annotated with `ComponentType`:

Example 74.7. Register a custom source set

MarkdownPlugin.groovy

```
class MarkdownPlugin extends RuleSource {
    @ComponentType
    void registerMarkdownLanguage(TypeBuilder<MarkdownSourceSet> builder) {}
}
```

Setting the *language name* is mandatory.

74.5. Putting it all together

74.5.1. Generating binaries from components

Binaries generation from components is done via rules annotated with `ComponentBinaries`. This rule generates a `DocumentationBinary` named `exploded` for each `DocumentationComponent` and sets its `outputDir` property:

Example 74.8. Generates documentation binaries

DocumentationPlugin.groovy

```
class DocumentationPlugin extends RuleSource {
    @ComponentBinaries
    void generateDocBinaries(ModelMap<DocumentationBinary> binaries, VariantComp
        binaries.create("exploded") { binary ->
            outputDir = new File(buildDir, "${component.name}/${binary.name}")
        }
    }
}
```

74.5.2. Generating tasks from binaries

Tasks generation from binaries is done via rules annotated with `BinaryTasks`. This rule generates a `Copy` task for each `TextSourceSet` of each `DocumentationBinary`:

Example 74.9. Generates tasks for text source sets

DocumentationPlugin.groovy

```
class DocumentationPlugin extends RuleSource {
    @BinaryTasks
    void generateTextTasks(ModelMap<Task> tasks, final DocumentationBinary binary) {
        binary.inputs.withType(TextSourceSet) { textSourceSet ->
            def taskName = binary.tasks.taskName("compile", textSourceSet.name)
            def outputDir = new File(binary.outputDir, textSourceSet.name)
            tasks.create(taskName, Copy) {
                from textSourceSet.source
                destinationDir = outputDir
            }
        }
    }
}
```

This rule generates a MarkdownCompileTask task for each MarkdownSourceSet of each DocumentationBinary.

Example 74.10. Register a custom source set

MarkdownPlugin.groovy

```
class MarkdownPlugin extends RuleSource {
    @BinaryTasks
    void processMarkdownDocumentation(ModelMap<Task> tasks, final DocumentationBinary binary) {
        binary.inputs.withType(MarkdownSourceSet) { markdownSourceSet ->
            def taskName = binary.tasks.taskName("compile", markdownSourceSet.name)
            def outputDir = new File(binary.outputDir, markdownSourceSet.name)
            tasks.create(taskName, MarkdownHtmlCompile) { compileTask ->
                compileTask.source = markdownSourceSet.source
                compileTask.destinationDir = outputDir
                compileTask.smartQuotes = markdownSourceSet.smartQuotes
                compileTask.generateIndex = markdownSourceSet.generateIndex
            }
        }
    }
}
```

See the sample source for more on the MarkdownCompileTask task.

74.5.3. Using your custom model

This build script demonstrate usage of the custom model defined in the sections above:

Example 74.11. an example of using a custom software model

build.gradle

```
import sample.documentation.DocumentationComponent
import sample.documentation.TextSourceSet
import sample.markdown.MarkdownSourceSet

apply plugin:sample.documentation.DocumentationPlugin
apply plugin:sample.markdown.MarkdownPlugin

model {
    components {
        docs(DocumentationComponent) {
            sources {
                reference(TextSourceSet)
                userguide(MarkdownSourceSet) {
                    generateIndex = true
                    smartQuotes = true
                }
            }
        }
    }
}
```

Note: The code for this example can be found at `samples/customModel/languageType/` in the '-all' distribution of Gradle.

And in the components reports for such a build script we can see our model types properly registered:

Example 74.12. components report

Output of `gradle -q components`

```
> gradle -q components
```

```
-----
Root project
-----
```

```
DocumentationComponent 'docs'
-----
```

Source sets

```
    Markdown source 'docs:userguide'
        srcDir: src/docs/userguide
    Text source 'docs:reference'
        srcDir: src/docs/reference
```

Binaries

```
    DocumentationBinary 'docs:exploded'
        build using task: :docsExploded
```

Note: currently not all plugins register their components, so some components may nc

74.6. About internal views

Internal views can be added to an already registered type or to a new custom type. In other words, using internal views, you can attach extra properties to already registered components, binaries and source sets types like `JvmLibrarySpec`, `JarBinarySpec` or `JavaSourceSet` and to the custom types you write.

Let's start with a simple component public type and its internal view declarations:

Example 74.13. public type and internal view declaration

build.gradle

```
@Managed interface MyComponent extends ComponentSpec {
    String getPublicData()
    void setPublicData(String data)
}
@Managed interface MyComponentInternal extends MyComponent {
    String getInternalData()
    void setInternalData(String internal)
}
```

The type registration is as follows:

Example 74.14. type registration

build.gradle

```
class MyPlugin extends RuleSource {
    @ComponentType
    void registerMyComponent(TypeBuilder<MyComponent> builder) {
        builder.internalView(MyComponentInternal)
    }
}
```

The `internalView(type)` method of the type builder can be called several times. This is how you would add several internal views to a type.

Now, let's mutate both public and internal data using some rule:

Example 74.15. public and internal data mutation

build.gradle

```
class MyPlugin extends RuleSource {
    @Mutate
    void mutateMyComponents(ModelMap<MyComponentInternal> components) {
        components.all { component ->
            component.publicData = "Some PUBLIC data"
            component.internalData = "Some INTERNAL data"
        }
    }
}
```

Our `internalData` property should not be exposed to build logic. Let's check this using the `model` task on the following build file:

Example 74.16. example build script and model report output

build.gradle

```
apply plugin: MyPlugin
model {
    components {
        my(MyComponent)
    }
}
```

Output of `gradle -q model`

```
> gradle -q model

-----
Root project
-----

+ components
  | Type:      org.gradle.platform.base.ComponentSpecContainer
  | Creator:    ComponentBasePlugin.PluginRules#components(ComponentSpecContain
  | Rules:
    components { ... } @ build.gradle line 42, column 5
    MyPlugin#mutateMyComponents(ModelMap<MyComponentInternal>)
+ my
  | Type:      MyComponent
  | Creator:    components { ... } @ build.gradle line 42, column 5 > creat
  | Rules:
    MyPlugin#mutateMyComponents(ModelMap<MyComponentInternal>) > all()
+ publicData
  | Type:      java.lang.String
  | Value:      Some PUBLIC data
  | Creator:    components { ... } @ build.gradle line 42, column 5 > c
+ tasks
  | Type:      org.gradle.model.ModelMap<org.gradle.api.Task>
  | Creator:    Project.<init>.tasks()
+ assemble
  | Type:      org.gradle.api.DefaultTask
  | Value:      task ':assemble'
  | Creator:    tasks.addPlaceholderAction(assemble)
  | Rules:
    copyToTaskContainer
+ build
  | Type:      org.gradle.api.DefaultTask
  | Value:      task ':build'
  | Creator:    tasks.addPlaceholderAction(build)
  | Rules:
    copyToTaskContainer
+ buildEnvironment
  | Type:      org.gradle.api.tasks.diagnostics.BuildEnvironmentReportTask
  | Value:      task ':buildEnvironment'
  | Creator:    tasks.addPlaceholderAction(buildEnvironment)
  | Rules:
    copyToTaskContainer
+ check
  | Type:      org.gradle.api.DefaultTask
```

```

        | Value:      task ':check'
        | Creator:    tasks.addPlaceholderAction(check)
        | Rules:
            copyToTaskContainer
+ clean
    | Type:      org.gradle.api.tasks.Delete
    | Value:      task ':clean'
    | Creator:    tasks.addPlaceholderAction(clean)
    | Rules:
        copyToTaskContainer
+ components
    | Type:      org.gradle.api.reporting.components.ComponentReport
    | Value:      task ':components'
    | Creator:    tasks.addPlaceholderAction(components)
    | Rules:
        copyToTaskContainer
+ dependencies
    | Type:      org.gradle.api.tasks.diagnostics.DependencyReportTask
    | Value:      task ':dependencies'
    | Creator:    tasks.addPlaceholderAction(dependencies)
    | Rules:
        copyToTaskContainer
+ dependencyInsight
    | Type:      org.gradle.api.tasks.diagnostics.DependencyInsightReportTask
    | Value:      task ':dependencyInsight'
    | Creator:    tasks.addPlaceholderAction(dependencyInsight)
    | Rules:
        HelpTasksPlugin.Rules#addDefaultDependenciesReportConfiguration(Depend
        copyToTaskContainer
+ dependentComponents
    | Type:      org.gradle.api.reporting.dependents.DependentComponentsRepor
    | Value:      task ':dependentComponents'
    | Creator:    tasks.addPlaceholderAction(dependentComponents)
    | Rules:
        copyToTaskContainer
+ help
    | Type:      org.gradle.configuration.Help
    | Value:      task ':help'
    | Creator:    tasks.addPlaceholderAction(help)
    | Rules:
        copyToTaskContainer
+ init
    | Type:      org.gradle.buildinit.tasks.InitBuild
    | Value:      task ':init'
    | Creator:    tasks.addPlaceholderAction(init)
    | Rules:
        copyToTaskContainer
+ model
    | Type:      org.gradle.api.reporting.model.ModelReport
    | Value:      task ':model'
    | Creator:    tasks.addPlaceholderAction(model)
    | Rules:
        copyToTaskContainer
+ projects
    | Type:      org.gradle.api.tasks.diagnostics.ProjectReportTask
    | Value:      task ':projects'
    | Creator:    tasks.addPlaceholderAction(projects)
    | Rules:
        copyToTaskContainer
+ properties
    | Type:      org.gradle.api.tasks.diagnostics.PropertyReportTask

```

```

    | Value:      task ':properties'
    | Creator:    tasks.addPlaceholderAction(properties)
    | Rules:
      copyToTaskContainer
+ tasks
    | Type:      org.gradle.api.tasks.diagnostics.TaskReportTask
    | Value:      task ':tasks'
    | Creator:    tasks.addPlaceholderAction(tasks)
    | Rules:
      copyToTaskContainer
+ wrapper
    | Type:      org.gradle.api.tasks.wrapper.Wrapper
    | Value:      task ':wrapper'
    | Creator:    tasks.addPlaceholderAction(wrapper)

```

```
| Rules:
  copyToTaskContainer
```

We can see in this report that `publicData` is present and that `internalData` is not.

Part VII. Appendix

A

Gradle Samples

Listed below are some of the stand-alone samples which are included in the Gradle distribution. You can find these samples in the `GRADLE_HOME/samples` directory of the distribution.

Table A.1. Samples included in the distribution

Sample	Description
announce	A project which uses the announce plugin
application	A project which uses the application plugin
buildDashboard	A project which uses the build-dashboard plugin
codeQuality	A project which uses the various code quality plugins.
customBuildLanguage	This sample demonstrates how to add some custom elements to the build DSL. It also demonstrates the use of custom plug-ins to organize build logic.
customDistribution	This sample demonstrates how to create a custom Gradle distribution and use it with the Gradle wrapper.
customPlugin	A set of projects that show how to implement, test, publish and use a custom plugin and task.
ear/earCustomized/ear	Web application ear project with customized contents
ear/earWithWar	Web application ear project
groovy/crossCompilation	A project doing cross compilation for a Groovy Project to Java 6

<code>groovy/customizedLayout</code>	Groovy project with a custom source layout
<code>groovy/mixedJavaAndGroovy</code>	Project containing a mix of Java and Groovy source
<code>groovy/multiproject</code>	Build made up of multiple Groovy projects. Also demonstrates how to exclude certain source files, and the use of a custom Groovy AST transformation.
<code>groovy/quickstart</code>	Groovy quickstart sample
<code>java-library/multiproject</code>	Java Library multiproject
<code>java-library/quickstart</code>	Java Library quickstart project
<code>java/base</code>	Java base project
<code>java/crossCompilation</code>	A project doing cross compilation to Java 6
<code>java/customizedLayout</code>	Java project with a custom source layout
<code>java/multiproject</code>	This sample demonstrates how an application can be composed using multiple Java projects.
<code>java/quickstart</code>	Java quickstart project
<code>java/withIntegrationTests</code>	This sample demonstrates how to use a source set to add an integration test suite to a Java project.
<code>javaGradlePlugin</code>	This example demonstrates the use of the java gradle plugin development plugin. By applying the plugin, the java plugin is automatically applied as well as the <code>gradleApi()</code> dependency. Furthermore, validations are performed against the plugin metadata during jar execution.
<code>maven/pomGeneration</code>	Demonstrates how to deploy and install to a Maven repository. Also demonstrates how to deploy a javadoc JAR along with the main JAR, how to customize the contents of the generated POM, and how to deploy snapshots and releases to different repositories.

maven/quickstart	Demonstrates how to deploy and install artifacts to a Maven repository.
osgi	A project which builds an OSGi bundle
plugins	A set of projects that show how to implement, test, publish and use a custom plugins with the latest technology.
scala/crossCompilation	A project doing cross compilation for a Scala project to Java 6
scala/customizedLayout	Scala project with a custom source layout
scala/force	Scala quickstart project
scala/mixedJavaAndScala	A project containing a mix of Java and Scala source.
scala/quickstart	Scala quickstart project
scala/zinc	Scala project using the Zinc based Scala compiler.
testing/testReport	Generates an HTML test report that includes the test results from all subprojects.
toolingApi/customModel	A sample of how a plugin can expose its own custom tooling model to tooling API clients.
toolingApi/eclipse	An application that uses the tooling API to build the Eclipse model for a project.
toolingApi/idea	An application that uses the tooling API to extract information needed by IntelliJ IDEA.
toolingApi/model	An application that uses the tooling API to build the model for a Gradle build.
toolingApi/runBuild	An application that uses the tooling API to run a Gradle task.
userguide/distribution	A project which uses the distribution plugin

<code>userguide/javaLibraryDistribution</code>	A project which uses the Java library distribution plugin
<code>webApplication/customized</code>	Web application with customized WAR contents.
<code>webApplication/quickstart</code>	Web application quickstart project

A.1. Sample `customBuildLanguage`

This sample demonstrates how to add some custom elements to the build DSL. It also demonstrates the use of custom plug-ins to organize build logic.

The build is composed of 2 types of projects. The first type of project represents a product, and the second represents a product module. Each product includes one or more product modules, and each product module may be included in multiple products. That is, there is a many-to-many relationship between these products and product modules. For each product, the build produces a ZIP containing the runtime classpath for each product module included in the product. The ZIP also contains some product-specific files.

The custom elements can be seen in the build script for the product projects (for example, `basicEdition/build.gradle`). Notice that the build script uses the `product { }` element. This is a custom element.

The build scripts of each project contain only declarative elements. The bulk of the work is done by 2 custom plug-ins found in `buildSrc/src/main/groovy`.

A.2. Sample `customDistribution`

This sample demonstrates how to create a custom Gradle distribution and use it with the Gradle wrapper.

This sample contains the following projects:

- The `plugin` directory contains the project that implements a custom plugin, and bundles the plugin into a custom Gradle distribution.
- The `consumer` directory contains the project that uses the custom distribution.

A.3. Sample `customPlugin`

A set of projects that show how to implement, test, publish and use a custom plugin and task.

This sample contains the following projects:

- The `plugin` directory contains the project that implements and publishes the plugin.
- The `consumer` directory contains the project that uses the plugin.

A.4. Sample java/multiproject

This sample demonstrates how an application can be composed using multiple Java projects.

This build creates a client-server application which is distributed as 2 archives. First, there is a client ZIP which includes an API JAR, which a 3rd party application would compile against, and a client runtime. Then, there is a server WAR which provides a web service.

A.5. Sample plugins

A set of projects that show how to implement, test, publish and use a custom plugins with the latest technology.

This sample contains the following projects:

- The `buildscript` directory contains a project that uses the old `buildscript` syntax for using plugins.
- The `dsl` directory contains the a project that uses the new `plugins` syntax for using plugins.
- The `publishing` directory contains a complete example of the modern publishing plugins working with the `java-gradle-plugin` to produce two plugins shipped in the same jar and being published to both an ivy and maven repository.
- The `consuming` directory contains an example of resolving plugins from custom repositories instead the Gradle Plugin Portal.

B

Potential Traps

B.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 B.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. ^[31]

B.2. Configuration and execution phase

It is important to keep in mind that Gradle has a distinct configuration and execution phase (see Chapter 22, *The Build Lifecycle*).

Example B.2. Distinct configuration and execution phase

build.gradle

```
def classesDir = file('build/classes')
classesDir.mkdirs()
task clean(type: Delete) {
    delete 'build'
}
task compile(dependsOn: 'clean') {
    doLast {
        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.

[31] One of those discussions can be found here:
<http://groovy.329449.n5.nabble.com/script-scoping-question-td355887.html>

The Feature Lifecycle

Gradle is under constant development and improvement. New versions are delivered on a regular and frequent basis (approximately every 6 weeks). Continuous improvement combined with frequent delivery allows new features to be made available to users early and for invaluable real world feedback to be incorporated into the development process. Getting new functionality into the hands of users regularly is a core value of the Gradle platform. At the same time, API and feature stability is taken very seriously and is also considered a core value of the Gradle platform. This is something that is engineered into the development process by design choices and automated testing, and is formalised by Section C.2, “Backwards Compatibility Policy”.

The Gradle *feature lifecycle* has been designed to meet these goals. It also serves to clearly communicate to users of Gradle what the state of a feature is. The term *feature* typically means an API or DSL method or property in this context, but it is not restricted to this definition. Command line arguments and modes of execution (e.g. the Build Daemon) are two examples of other kinds of features.

C.1. States

Features can be in one of 4 states:

- Internal
- Incubating
- Public
- Deprecated

C.1.1. Internal

Internal features are not designed for public use and are only intended to be used by Gradle itself. They can change in any way at any point in time without any notice. Therefore, we recommend avoiding the use of such features. Internal features are not documented. If it appears in this User Guide, the DSL Reference or the API Reference documentation then the feature is not internal.

Internal features may evolve into public features.

C.1.2. Incubating

Features are introduced in the *incubating* state to allow real world feedback to be incorporated into the feature before it is made public and locked down to provide backwards compatibility. It also gives users who are willing to accept potential future changes early access to the feature so they can put it into use immediately.

A feature in an incubating state may change in future Gradle versions until it is no longer incubating. Changes to incubating features for a Gradle release will be highlighted in the release notes for that release. The incubation period for new features varies depending on the scope, complexity and nature of the feature.

Features in incubation are clearly indicated to be so. In the source code, all methods/properties/classes that are incubating are annotated with `Incubating`, which is also used to specially mark them in the DSL and API references. If an incubating feature is discussed in this User Guide, it will be explicitly said to be in the incubating state.

C.1.3. Public

The default state for a non-internal feature is public. Anything that is documented in the User Guide, DSL Reference or API references that is not explicitly said to be incubating or deprecated is considered public. Features are said to be promoted from an incubating state to public. The release notes for each release indicate which previously incubating features are being promoted by the release.

A public feature will never be removed or intentionally changed without undergoing deprecation. All public features are subject to the backwards compatibility policy.

C.1.4. Deprecated

Some features will become superseded or irrelevant due to the natural evolution of Gradle. Such features will eventually be removed from Gradle after being deprecated. A deprecated feature will never be changed, until it is finally removed according to the backwards compatibility policy.

Deprecated features are clearly indicated to be so. In the source code, all methods/properties/classes that are deprecated are annotated with “`@java.lang.Deprecated`” which is reflected in the DSL and API references. In most cases, there is a replacement for the deprecated element, and this will be described in the documentation. Using a deprecated feature will also result in a runtime warning in Gradle's output.

Use of deprecated features should be avoided. The release notes for each release indicate any features that are being deprecated by the release.

C.2. Backwards Compatibility Policy

Gradle provides backwards compatibility across major versions (e.g. 1.x, 2.x, etc.). Once a public feature is introduced or promoted in a Gradle release it will remain indefinitely or until it is deprecated. Once deprecated, it may be removed in the next major release. Deprecated features may be supported across major releases, but this is not guaranteed.

D

Gradle Command Line

The **gradle** command has the following usage:

```
gradle [option...] [task...]
```

The command-line options available for the **gradle** command are listed below:

-?, -h, --help

Shows a help message.

-a, --no-rebuild

Do not rebuild project dependencies.

--all

Shows additional detail in the task listing. See Section 4.7.2, “Listing tasks”.

-b, --build-file

Specifies the build file. See Section 4.5, “Selecting which build to execute”.

-c, --settings-file

Specifies the settings file.

--console

Specifies which type of console output to generate.

Set to `plain` to generate plain text only. This option disables all color and other rich output in the console output.

Set to `auto` (the default) to enable color and other rich output in the console output when the build process is attached to a console, or to generate plain text only when not attached to a console.

Set to `rich` to enable color and other rich output in the console output, regardless of whether the build process is not attached to a console. When not attached to a console, the build output will use ANSI control characters to generate the rich output.

--continue

Continues task execution after a task failure.

--configure-on-demand (incubating)

Only relevant projects are configured in this build run. This means faster builds for large multi-projects. See the section called “Configuration on demand”.

-D, --system-prop

Sets a system property of the JVM, for example `-Dmyprop=myvalue`. See Section 12.2, “Gradle properties and system properties”.

-d, --debug

Log in debug mode (includes normal stacktrace). See Chapter 24, *Logging*.

-g, --gradle-user-home

Specifies the Gradle user home directory. The default is the `.gradle` directory in the user's home directory.

--gui

Launches the Gradle GUI. See Chapter 11, *Using the Gradle Graphical User Interface*.

The Gradle GUI has been deprecated and will be removed in Gradle 4.0. Consider using an IDE with support for Gradle e.g. Eclipse, IntelliJ or NetBeans instead.

--include-build

Run the build as a composite, including the specified build. See Chapter 10, *Composite builds*.

-I, --init-script

Specifies an initialization script. See Chapter 44, *Initialization Scripts*.

-i, --info

Set log level to info. See Chapter 24, *Logging*.

-m, --dry-run

Runs the build with all task actions disabled. See Section 4.8, “Dry Run”.

--offline

Specifies that the build should operate without accessing network resources. See Section 25.9.2, “Command line options to override caching”.

-P, --project-prop

Sets a project property of the root project, for example `-Pmyprop=myvalue`. See Section 12.2, “Gradle properties and system properties”.

-p, --project-dir

Specifies the start directory for Gradle. Defaults to current directory. See Section 4.5, “Selecting which build to execute”.

--parallel (incubating)

Build projects in parallel. Gradle will attempt to determine the optimal number of executor threads to use. This option should only be used with decoupled projects (see Section 26.9, “Decoupled Projects”). For limitations of this option please see Section 26.8, “Parallel project execution”.

--max-workers (incubating)

Sets the maximum number of workers that Gradle may use. For example `--max-workers=3`. The default is the number of processors.

--profile

Profiles build execution time and generates a report in the *buildDir/reports/profile* directory. See Section 4.7.8, “Profiling a build”.

--project-cache-dir

Specifies the project-specific cache directory. Default value is *.gradle* in the root project directory.

-q, --quiet

Log errors only. See Chapter 24, *Logging*.

--recompile-scripts

Forces scripts to be recompiled, bypassing caching.

--refresh-dependencies

Refresh the state of dependencies. See Section 25.9.2, “Command line options to override caching”.

--rerun-tasks

Specifies that any task optimization is ignored.

-S, --full-stacktrace

Print out the full (very verbose) stacktrace for any exceptions. See Chapter 24, *Logging*.

-s, --stacktrace

Print out the stacktrace also for user exceptions (e.g. compile error). See Chapter 24, *Logging*.

--scan (incubating)

Creates a build scan. Gradle will fail the build if the build scan plugin has not been applied. For more information about build scans, please visit <https://gradle.com>.

--no-scan (incubating)

Disables the creation of a build scan. For more information about build scans, please visit <https://gradle.com>.

-t, --continuous (incubating)

Enables continuous building - Gradle will automatically re-run when changes are detected.

-u, --no-search-upward

Don't search in parent directories for a *settings.gradle* file.

-v, --version

Prints version info.

-x, --exclude-task

Specifies a task to be excluded from execution. See Section 4.2, “Excluding tasks”.

The above information is printed to the console when you execute **gradle -h**.

D.1. Daemon command-line options

The Chapter 6, *The Gradle Daemon* contains more information about the daemon. For example it includes information how to turn on the daemon by default so that you can avoid using `--daemon` all the time.

--daemon

Uses the Gradle daemon to run the build. Starts the daemon if not running or existing daemon busy. Chapter 6, *The Gradle Daemon* contains more detailed information when new daemon processes are started.

--foreground

Starts the Gradle daemon in the foreground. Useful for debugging or troubleshooting because you can easily monitor the build execution.

--no-daemon

Do not use the Gradle daemon to run the build. Useful occasionally if you have configured Gradle to always run with the daemon by default.

--stop

Stops the Gradle daemon if it is running. You can only stop daemons that were started with the Gradle version you use when running `--stop`.

D.2. System properties

The following system properties are available for the **gradle** command. Note that command-line options take precedence over system properties.

`gradle.user.home`

Specifies the Gradle user home directory.

The Section 12.1, “Configuring the build environment via `gradle.properties`” contains specific information about Gradle configuration available via system properties.

D.3. Environment variables

The following environment variables are available for the **gradle** command. Note that command-line options and system properties take precedence over environment variables.

GRADLE_OPTS

Specifies command-line arguments to use to start the JVM. This can be useful for setting the system properties to use for running Gradle. For example you could set `GRADLE_OPTS="-Dorg.gradle.daemon"` to use the Gradle daemon without needing to use the `--daemon` option every time you run Gradle.

Section 12.1, “Configuring the build environment via `gradle.properties`” contains more information about ways of configuring the daemon without using environmental variables, e.g. in more maintainable and explicit way.

GRADLE_USER_HOME

Specifies the Gradle user home directory (which defaults to “`USER_HOME/.gradle`” if not set).

JAVA_HOME

Specifies the JDK installation directory to use.

E

Documentation licenses

E.1. Gradle Documentation

Copyright © 2007-2016 Gradle, Inc.

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.

E.2. Header link icon

Copyright © 2011–2013 VisualEditor team.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

The Software is provided "as is", without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the Software or the use or other dealings in the Software.

VII

Appendix

A

Artifact

??

B

Build Script

??

C

Configuration

See Dependency Configuration.

Configuration Injection

??

D

DAG

See Directed Acyclic Graph.

Dependency

See External Dependency.

See Project Dependency.

??

Dependency Configuration

??

Dependency Resolution

??

Directed Acyclic Graph

A directed acyclic graph is a directed graph that contains no cycles. In Gradle each task to execute represents a node in the graph. A `dependsOn` relation to another task will add this other task as a node (if it is not in the graph already) and create a directed edge between those two nodes. Any `dependsOn` relation will be validated for cycles. There must be no way to start at certain node, follow a sequence of edges and end up at the original node.

Domain Specific Language

A domain-specific language is a programming language or specification language dedicated to a particular problem domain, a particular problem representation technique, and/or a particular solution technique. The concept isn't new—special-purpose programming languages and all kinds of modeling/specification languages have always existed, but the term has become more popular due to the rise of domain-specific modeling.

DSL

See Domain Specific Language.

E

External Dependency

??

Extension Object

??

I

Init Script

A script that is run before the build itself starts, to allow customization of Gradle and the build.

Initialization Script

See Init Script.

P

Plugin

??

Project

??

Project Dependency

??

Publication

??

R

Repository

??

S

Source Set

??

T

Task

??

Transitive Dependency

??