Gradle User Guide

Version 2.3-rc-3

Table of Contents

List of Examples

8.3. Shortcut definition of an external dependency
8.4. Usage of Maven central repository
8.5. Usage of a remote Maven repository
8.6. Usage of a remote Ivy directory
8.7. Usage of a local Ivy directory
8.8. Publishing to an Ivy repository
8.9. Publishing to a Maven repository
9.1. Groovy plugin
9.2. Dependency on Groovy
9.3. Groovy example - complete build file
10.1. War plugin
10.2. Running web application with Jetty plugin
11.1. Executing multiple tasks
11.2. Excluding tasks
11.3. Abbreviated task name
11.4. Abbreviated camel case task name
11.5. Selecting the project using a build file
11.6. Selecting the project using project directory
11.7. Obtaining information about projects
11.8. Providing a description for a project
11.9. Obtaining information about tasks
11.10. Changing the content of the task report
11.11. Obtaining more information about tasks
11.12. Obtaining detailed help for tasks
11.13. Obtaining information about dependencies
11.14. Filtering dependency report by configuration
11.15. Getting the insight into a particular dependency
11.16. Information about properties
12.1. Launching the GUI
13.1. Accessing property of the Project object
13.2. Using local variables
13.3. Using extra properties
13.4. Groovy JDK methods
13.5. Property accessors
13.6. Method call without parentheses
13.7. List and map literals
13.8. Closure as method parameter
13.9. Closure delegates
14.1. Directory creation with mkdir
14.2. Setting properties with a gradle.properties file
14.3. Configuring the project using an external build script
14.4. Configuring arbitrary objects
14.5. Configuring arbitrary objects using a script
15.1. Defining tasks
15.2. Defining tasks - using strings for task names
15.3. Defining tasks with alternative syntax
15.4. Accessing tasks as properties
15.5. Accessing tasks via tasks collection

55.27. Defining tool chains
55.28. Reconfigure tool arguments
55.29. Defining target platforms
55.30. Registering CUnit tests
55.31. Registering CUnit tests
55.32. Running CUnit tests
56.1. Single project build
56.2. Hierarchical layout
56.3. Flat layout
56.4. Modification of elements of the project tree
56.5. Modification of elements of the project tree
56.6. Adding of test task to each project which has certain property set
56.7. Notifications
56.8. Setting of certain property to all tasks
56.9. Logging of start and end of each task execution
57.1. Multi-project tree - water & bluewhale projects
57.2. Build script of water (parent) project
57.3. Multi-project tree - water, bluewhale & krill projects
57.4. Water project build script
57.5. Defining common behavior of all projects and subprojects
57.6. Defining specific behaviour for particular project
57.7. Defining specific behaviour for project krill
57.8. Adding custom behaviour to some projects (filtered by project name)
57.9. Adding custom behaviour to some projects (filtered by project properties)
57.10. Running build from subproject
57.11. Evaluation and execution of projects
57.12. Evaluation and execution of projects
57.13. Running tasks by their absolute path
57.14. Dependencies and execution order
57.15. Dependencies and execution order
57.16. Dependencies and execution order
57.17. Declaring dependencies
57.18. Declaring dependencies
57.19. Cross project task dependencies
57.20. Configuration time dependencies
57.21. Configuration time dependencies - evaluationDependsOn
57.22. Configuration time dependencies
57.23. Dependencies - real life example - crossproject configuration
57.24. Project lib dependencies
57.25. Project lib dependencies
57.26. Fine grained control over dependencies
57.27. Build and Test Single Project
57.28. Partial Build and Test Single Project
57.29. Build and Test Depended On Projects
57.30. Build and Test Dependent Projects
58.1. Defining a custom task
58.2. A hello world task
58.3. A customizable hello world task

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.` 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

Overview

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.

3
Tutorials

3.1. Getting Started

The following tutorials introduce some of the basics of Gradle, to help you get started.

Chapter 4, *Installing Gradle*
: Describes how to install Gradle.

Chapter 6, *Build Script Basics*
: Introduces the basic build script elements: *projects* and *tasks*.

Chapter 7, *Java Quickstart*
: Shows how to start using Gradle's build-by-convention support for Java projects.

Chapter 8, *Dependency Management Basics*
: Shows how to start using Gradle's dependency management.

Chapter 9, *Groovy Quickstart*
: Using Gradle's build-by-convention support for Groovy projects.

Chapter 10, *Web Application Quickstart*
: Using Gradle's build-by-convention support for Web applications.

4
Installing Gradle

4.1. Prerequisites

Gradle requires a Java JDK or JRE to be installed, version 6 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.

4.2. Download

You can download one of the Gradle distributions from the Gradle web site.

4.3. 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 and Groovydoc).
- 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.

4.4. Environment variables

For running Gradle, add *`GRADLE_HOME`*`/bin` to your `PATH` environment variable. Usually, this is sufficient to run Gradle.

4.5. 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.

4.6. 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.

5
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.

5.1. 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 19, *The Gradle Daemon*.

5.2. 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.

6
Build Script Basics

6.1. 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.

6.2. 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. [2] 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 6.1. 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 6.2. 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 18, *Logging* for more details about the command-line options which affect Gradle's output.

6.3. A shortcut task definition

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

Example 6.3. A task definition shortcut

`build.gradle`

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

Again, this defines a task called `hello` with a single closure to execute. We will use this task definition style throughout the user guide.

6.4. Build scripts are code

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

Example 6.4. Using Groovy in Gradle's tasks

`build.gradle`

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

Output of **`gradle -q upper`**

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

or

Example 6.5. Using Groovy in Gradle's tasks

`build.gradle`

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

Output of **`gradle -q count`**

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

6.5. Task dependencies

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

Example 6.6. Declaration of task that depends on other task

`build.gradle`

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

Output of **`gradle -q intro`**

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

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

Example 6.7. Lazy dependsOn - the other task does not exist (yet)

`build.gradle`

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

Output of **`gradle -q taskX`**

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

The dependency of `taskX` to `taskY` is declared before `taskY` is defined. This is very important for multi-project builds. Task dependencies are discussed in more detail in Section 15.4, “Adding dependencies to a task”.

Please notice that you can't use shortcut notation (see Section 6.8, “Shortcut notations”) when referring to a task that is not yet defined.

6.6. 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 6.8. Dynamic creation of a task

`build.gradle`

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

Output of **`gradle -q task1`**

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

6.7. 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 6.9. Accessing a task via API - adding a dependency

`build.gradle`

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

Output of **`gradle -q task0`**

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

Or you can add behavior to an existing task.

Example 6.10. Accessing a task via API - adding behaviour

`build.gradle`

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

Output of **`gradle -q hello`**

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

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

6.8. Shortcut notations

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

Example 6.11. Accessing task as a property of the build script

`build.gradle`

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

Output of **`gradle -q hello`**

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

This enables very readable code, especially when using the tasks provided by the plugins, like the `compile` task.

6.9. 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 6.12. Adding extra properties to a task

`build.gradle`

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

task printTaskProperties << {
    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 13.4.2, “Extra properties”.

6.10. 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 6.13. Using AntBuilder to execute ant.loadfile target

`build.gradle`

```
task loadfile << {
    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 17, *Using Ant from Gradle*.

6.11. 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 6.14. Using methods to organize your build logic

`build.gradle`

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

task loadfile << {
    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 60, *Organizing Build Logic*.

6.12. Default tasks

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

Example 6.15. Defining a default tasks

`build.gradle`

```
defaultTasks 'clean', 'run'

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

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

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

Output of **`gradle -q`**

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

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

6.13. Configure by DAG

As we later describe in full detail (see Chapter 56, *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 6.16. Different outcomes of build depending on chosen tasks

`build.gradle`

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

task release(dependsOn: 'distribution') << {
    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).

6.14. 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 15, *More about Tasks*.

Otherwise, continue on to the tutorials in Chapter 7, *Java Quickstart* and Chapter 8, *Dependency Management Basics*.

---

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

7

Java Quickstart

7.1. 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.

7.2. A basic Java project

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

Example 7.1. 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.

7.2.1. 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 7.2. 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 `checkst`

plugin, this task will also run Checkstyle against your source code.

7.2.2. 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 7.3. 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 7.4. 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 8, *Dependency Management Basics*.

7.2.3. 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 7.5. Customization of MANIFEST.MF

build.gradle

```
sourceCompatibility = 1.5
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 7.6. 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.

7.2.4. 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 7.7. Publishing the JAR file

build.gradle

```
uploadArchives {
    repositories {
       flatDir {
           dirs 'repos'
       }
    }
}
```

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

7.2.5. Creating an Eclipse project

To create the Eclipse-specific descriptor files, like `.project`, you need to add another plugin to your build file:

Example 7.8. 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 38, *The Eclipse Plugins*.

7.2.6. Summary

Here's the complete build file for our sample:

Example 7.9. Java example - complete build file

`build.gradle`

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

sourceCompatibility = 1.5
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'
       }
    }
}
```

7.3. Multi-project Java build

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

Example 7.10. 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.

7.3.1. 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 7.11. 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 57, *Multi-project Builds*.

7.3.2. 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 7.12. Multi-project build - common configuration

`build.gradle`

```
subprojects {
    apply plugin: 'java'
    apply plugin: 'eclipse-wtp'

    repositories {
       mavenCentral()
    }

    dependencies {
        testCompile 'junit:junit:4.11'
    }

    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.

7.3.3. 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 7.13. Multi-project build - dependencies between projects

`api/build.gradle`

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

See Section 57.7.1, “Disabling the build of dependency projects” for how to disable this functionality.

7.3.4. Creating a distribution

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

Example 7.14. 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
}
```

7.4. 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 23, *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 8, *Dependency Management Basics*.

8

Dependency Management Basics

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

8.1. 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*.

8.2. Declaring your dependencies

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

Example 8.1. 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 >= 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.

8.3. 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 23.5, “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 51.3, “Dependency configurations” for the details of defining and customizing dependency configurations.

8.4. External dependencies

There are various types of dependencies that you can declare. One such type is an *external dependency*. This 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 8.2. 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 8.3. 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 51.4, “How to declare your dependencies”.

8.5. 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 8.4. Usage of Maven central repository

build.gradle

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

Or a remote Maven repository:

Example 8.5. Usage of a remote Maven repository

build.gradle

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

Or a remote Ivy repository:

Example 8.6. 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 8.7. 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 51.6, “Repositories”.

8.6. Publishing artifacts

Dependency configurations are also used to publish files.[3] 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 8.8. 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.[4] 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 8.9. 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 52, *Publishing artifacts*.

8.7. Where to next?

For all the details of dependency resolution, see Chapter 51, *Dependency Management*, and for artifact publication see Chapter 52, *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.

---

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

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

9

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 7, *Java Quickstart*.

9.1. A basic Groovy project

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

Example 9.1. 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 9.2. Dependency on Groovy

build.gradle

```
repositories {
    mavenCentral()
}

dependencies {
    compile 'org.codehaus.groovy:groovy-all:2.3.6'
}
```

Here is our complete build file:

Example 9.3. Groovy example - complete build file

`build.gradle`

```
apply plugin: 'eclipse'
apply plugin: 'groovy'

repositories {
    mavenCentral()
}

dependencies {
    compile 'org.codehaus.groovy:groovy-all:2.3.6'
    testCompile 'junit:junit:4.11'
}
```

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

9.2. 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 24, *The Groovy Plugin*, and you can find more sample Groovy projects in the `samples/groovy` directory in the Gradle distribution.

10
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.

10.1. Building a WAR file

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

Example 10.1. 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.

10.2. 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 Groovy libraries

Example 10.2. Running web application with Jetty plugin

build.gradle

```
apply plugin: 'jetty'
```

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.

10.3. Summary

You can find out more about the War plugin in Chapter 26, *The War Plugin* and the Jetty plugin in Chapter 28, *The Jetty Plugin*. You can find more sample Java projects in the `samples/webApplication` directory in the Gradle distribution.

11
Using the Gradle Command-Line

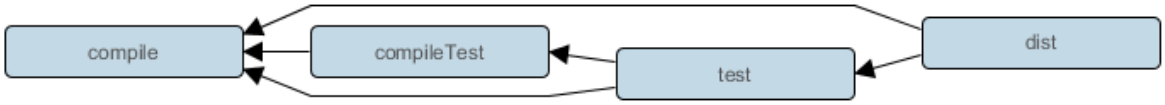
This chapter introduces the basics of the Gradle command-line. You run a build using the **gradle** command, which you have already seen in action in previous chapters.

11.1. 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 11.1. Task dependencies



Example 11.1. Executing multiple tasks

`build.gradle`

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

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

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

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

Output of **`gradle dist test`**

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

BUILD SUCCESSFUL

Total time: 1 secs
```

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

11.2. 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 11.2. 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.

11.3. 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).

11.4. 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 11.3. 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 11.4. 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.

11.5. 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 11.5. Selecting the project using a build file

`subdir/myproject.gradle`

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

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

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

Alternatively, you can use the `-p` option to specify the project directory to use. For multi-project builds you should use `-p` option instead of `-b` option.

Example 11.6. 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'.
```

11.6. 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.

11.6.1. 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 11.7. 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 11.8. Providing a description for a project

`build.gradle`

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

11.6.2. 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 11.9. 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
----------
components - Displays the components produced by root project 'projectReports'. [incu
dependencies - Displays all dependencies declared in root project 'projectReports'.
dependencyInsight - Displays the insight into a specific dependency in root project '
help - Displays a help message.
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 d

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. 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 11.10. 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, grouped by main task, and the dependencies for each task. Here is an example:

Example 11.11. 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, webapp:libs]
    docs - Builds the documentation
api:libs - Builds the JAR
    api:compile - Compiles the source files
webapp:libs - Builds the JAR [api:libs]
    webapp:compile - Compiles the source files

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

Help tasks
----------
components - Displays the components produced by root project 'projectReports'. [incu
api:components - Displays the components produced by project ':api'. [incubating]
webapp:components - Displays the components produced by project ':webapp'. [incubatin
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 project
help - Displays a help message.
api:help - Displays a help message.
webapp:help - Displays a help message.
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 d
api:tasks - Displays the tasks runnable from project ':api'.
webapp:tasks - Displays the tasks runnable from project ':webapp'.
```

11.6.3. 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 multiproject build. Below is an example of this detailed information:

Example 11.12. 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.

11.6.4. 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 11.13. 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:2.3.6

testCompile
\--- junit:junit:4.11
     \--- org.hamcrest:hamcrest-core:1.3

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

compile
+--- project :api
|    \--- org.codehaus.groovy:groovy-all:2.3.6
\--- 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 11.14. 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:4.11
     \--- org.hamcrest:hamcrest-core:1.3
```

11.6.5. 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 11.15. 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:2.3.6
\--- 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 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 plugin is applied, the dependencyInsight task is pre-configured with '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 '--configuration' option. For more information see the `DependencyInsightReportTask` class in the API documentation.

11.6.6. 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 11.16. Information about properties

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

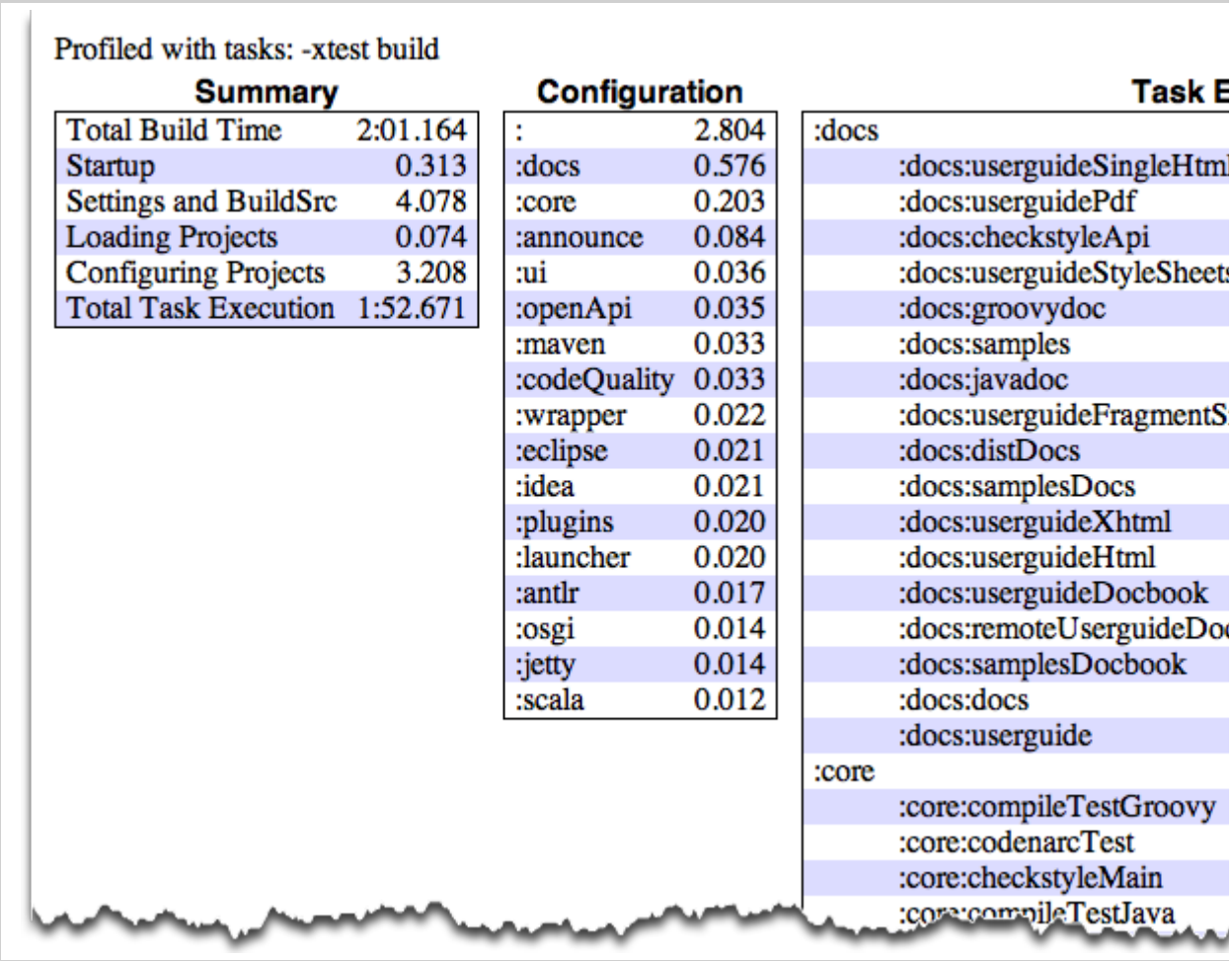
```
> gradle -q api:properties
------------------------------------------------------------
Project :api - The shared API for the application
------------------------------------------------------------

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@123
asDynamicObject: org.gradle.api.internal.ExtensibleDynamicObject@12345
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.grad
```

11.6.7. 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/buil` directory.



Profiled with tasks: -xtest build

Summary	
Total Build Time	2:01.164
Startup	0.313
Settings and BuildSrc	4.078
Loading Projects	0.074
Configuring Projects	3.208
Total Task Execution	1:52.671

Configuration	
:	2.804
:docs	0.576
:core	0.203
:announce	0.084
:ui	0.036
:openApi	0.035
:maven	0.033
:codeQuality	0.033
:wrapper	0.022
:eclipse	0.021
:idea	0.021
:plugins	0.020
:launcher	0.020
:antlr	0.017
:osgi	0.014
:jetty	0.014
:scala	0.012

Task E
:docs
:docs:userguideSingleHtml
:docs:userguidePdf
:docs:checkstyleApi
:docs:userguideStyleSheets
:docs:groovydoc
:docs:samples
:docs:javadoc
:docs:userguideFragmentS
:docs:distDocs
:docs:samplesDocs
:docs:userguideXhtml
:docs:userguideHtml
:docs:userguideDocbook
:docs:remoteUserguideDoc
:docs:samplesDocbook
:docs:docs
:docs:userguide
:core
:core:compileTestGroovy
:core:codenarcTest
:core:checkstyleMain
:core:compileTestJava

11.7. 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.

11.8. 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*.

12

Using the Gradle Graphical User Interface

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

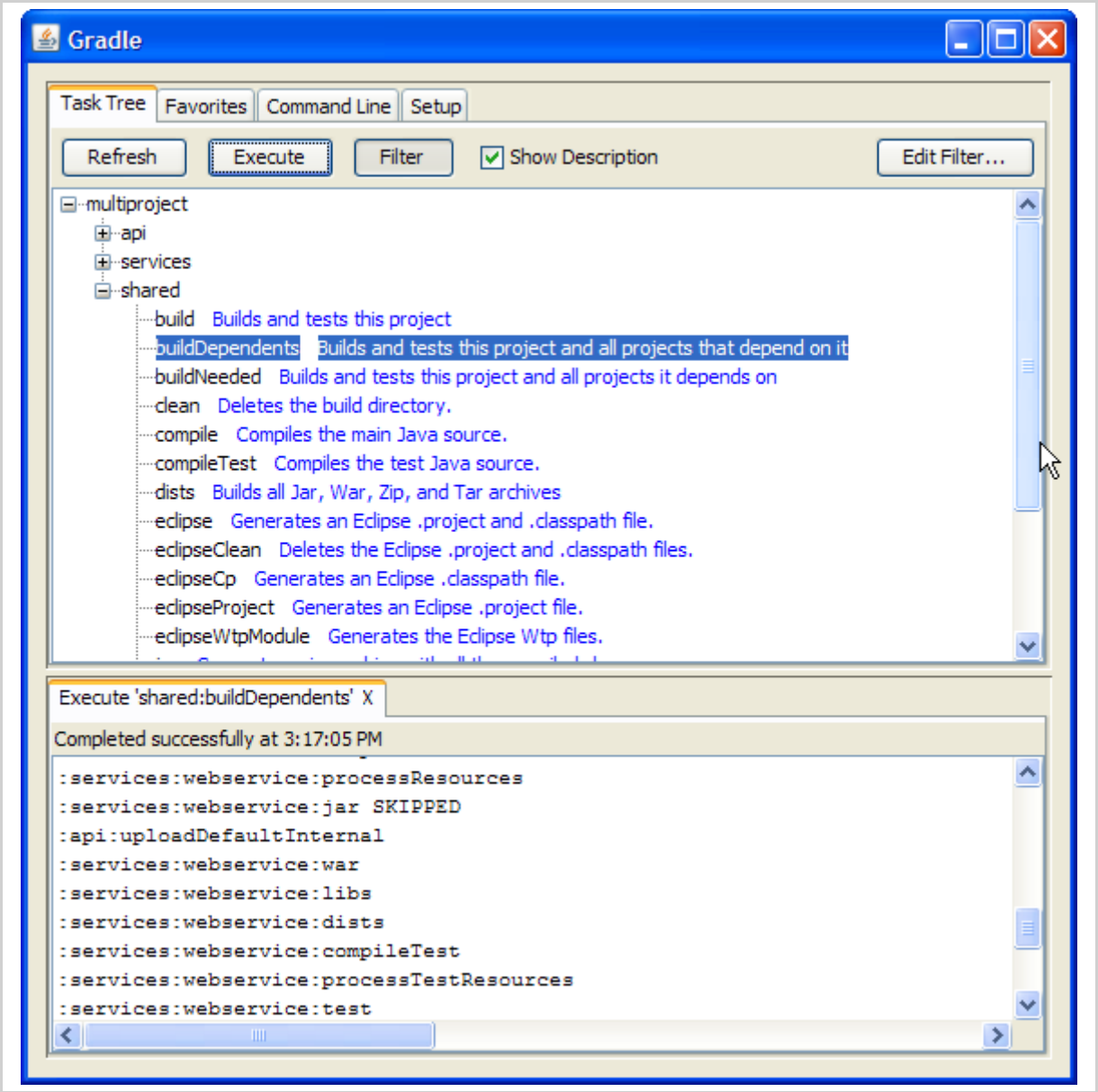
Example 12.1. Launching the GUI

```
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 12.1. 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.

12.1. 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.

12.2. 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 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.

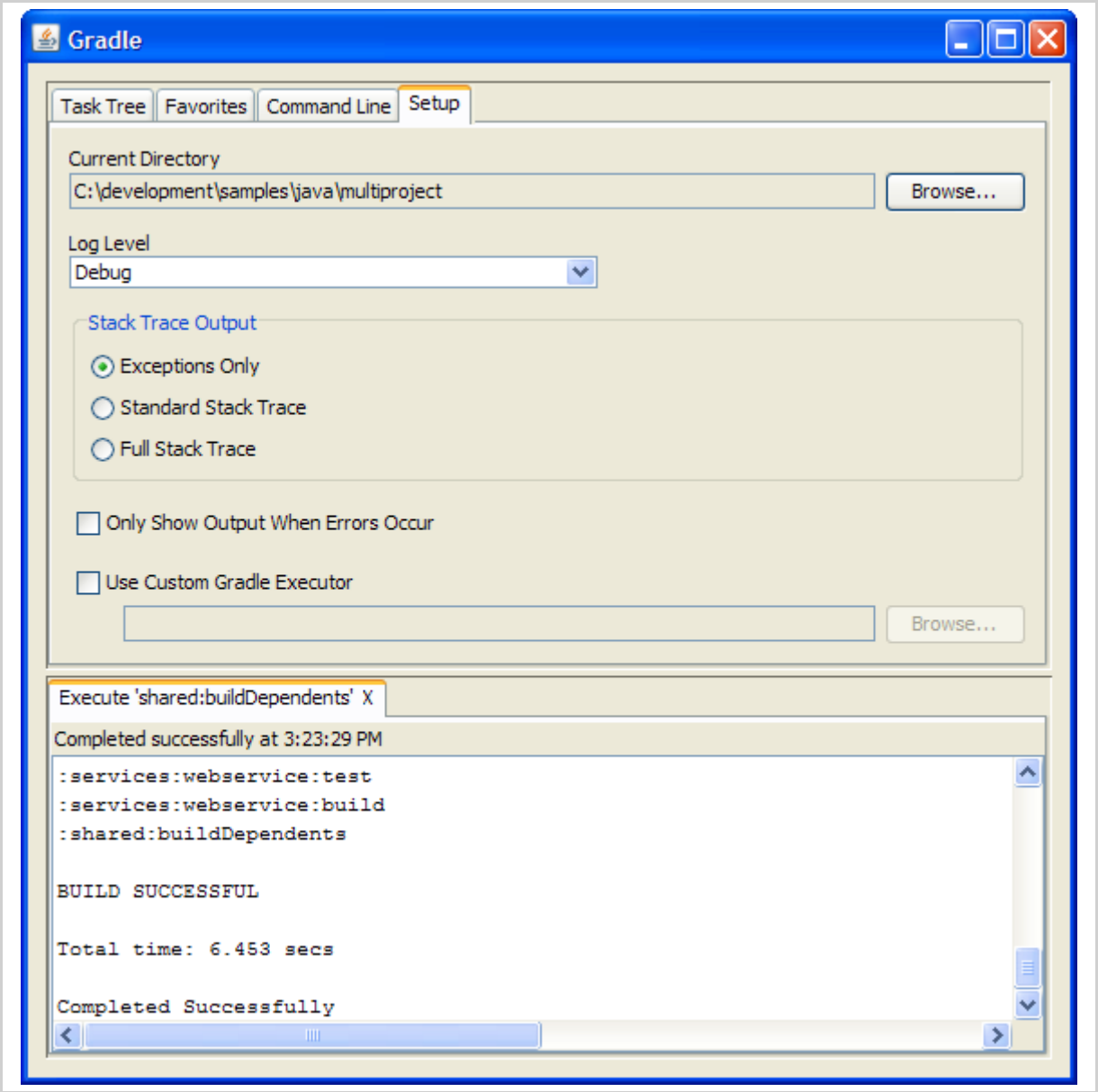
12.3. Command Line

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.

12.4. Setup

The Setup tab allows configuration of some general settings.

Figure 12.2. 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).

13
Writing Build Scripts

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

13.1. 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. [5] Gradle assumes that each build script is encoded using UTF-8.

13.2. The Project API

In the tutorial in Chapter 7, *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.

Example 13.1. Accessing property of the Project object

`build.gradle`

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

Output of **`gradle -q check`**

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

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.

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.

13.2.1. 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 13.1. Project Properties

Name	Type	Default Value
`project`	`Project`	The `Project` instance
`name`	`String`	The name of the project directory.
`path`	`String`	The absolute path of the project.
`description`	`String`	A description for the project.
`projectDir`	`File`	The directory containing the build script.
`buildDir`	`File`	*`projectDir`*`/build`
`group`	`Object`	`unspecified`
`version`	`Object`	`unspecified`
`ant`	`AntBuilder`	An `AntBuilder` instance

13.3. 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.

13.4. Declaring variables

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

13.4.1. 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 13.2. Using local variables

`build.gradle`

```
def dest = "dest"

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

13.4.2. 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 13.3. 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 << {
    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.

13.5. Some Groovy basics

Groovy 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.

13.5.1. 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 13.4. Groovy JDK methods

`build.gradle`

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

Have a look at http://groovy.codehaus.org/groovy-jdk/ for more details.

13.5.2. Property accessors

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

Example 13.5. Property accessors

`build.gradle`

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

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

13.5.3. Optional parentheses on method calls

Parentheses are optional for method calls.

Example 13.6. Method call without parentheses

build.gradle

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

13.5.4. 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 13.7. 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'
```

13.5.5. 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 13.8. Closure as method parameter

build.gradle

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

13.5.6. 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 13.9. Closure delegates

`build.gradle`

```
dependencies {
    assert delegate == project.dependencies
    testCompile('junit:junit:4.11')
    delegate.testCompile('junit:junit:4.11')
}
```

---

[5] Any language element except for statement labels.

14

Tutorial - 'This and That'

14.1. Directory creation

There is a common situation where multiple tasks depend on the existence of a directory. Of course you can deal with this by adding a `mkdir` to the beginning of those tasks, but it's almost always a bad idea to repeat a sequence of code that you only need once (Look up the *DRY* principle). A better solution would use the *dependsOn* relationship between tasks to reuse the task to create the directory:

Example 14.1. Directory creation with mkdir

`build.gradle`

```
def classesDir = new File('build/classes')

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

Output of **`gradle -q compile`**

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

14.2. 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. [6]

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 14.2. Setting properties with a gradle.properties file

`gradle.properties`

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

`build.gradle`

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

Output of **`gradle -q -PcommandLineProjectProp=commandLineProjectPropValue -Dorg.gr`**

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

14.2.1. 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`.

14.3. 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 14.3. Configuring the project using an external build script

`build.gradle`

```
apply from: 'other.gradle'
```

`other.gradle`

```
println "configuring $project"
task hello << {
    println 'hello from other script'
}
```

Output of **`gradle -q hello`**

```
> gradle -q hello
configuring root project 'configureProjectUsingScript'
hello from other script
```

14.4. Configuring arbitrary objects

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

Example 14.4. Configuring arbitrary objects

`build.gradle`

```
task configure << {
    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
```

14.5. Configuring arbitrary objects using an external script

You can also configure arbitrary objects using an external script.

Example 14.5. Configuring arbitrary objects using a script

`build.gradle`

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

`other.gradle`

```

```

Output of **`gradle -q configure`**

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

14.6. Caching

To improve responsiveness Gradle caches all compiled scripts by default. This includes all build scripts, initialization scripts, and other scripts. The first time you run a build for a project, Gradle creates a `.gradle` directory in which it puts the compiled script. The next time you run this build, Gradle uses the compiled script, if the script has not changed since it was compiled. Otherwise the script gets compiled and the new version is stored in the cache. If you run Gradle with the `--recompile-scripts` option, the cached script is discarded and the script is compiled and stored in the cache. This way you can force Gradle to rebuild the cache.

---

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

15
More about Tasks

In the introductory tutorial (Chapter 6, *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.

15.1. Defining tasks

We have already seen how to define tasks using a keyword style in Chapter 6, *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 15.1. Defining tasks

build.gradle

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

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

You can also use strings for the task names:

Example 15.2. Defining tasks - using strings for task names

build.gradle

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

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

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

Example 15.3. Defining tasks with alternative syntax

build.gradle

```
tasks.create(name: 'hello') << {
    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.

15.2. 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 15.4. Accessing tasks as properties

build.gradle

```
task hello

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

Tasks are also available through the `tasks` collection.

Example 15.5. 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 15.6. 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.

15.3. 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 15.7. 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 15.8. 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 15.9. 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 15.10. Defining a task with closure

build.gradle

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

15.4. Adding dependencies to a task

There are several ways you can define the dependencies of a task. In Section 6.5, “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`:

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 section of your task will get executed during the configuration phase of the build regardless of what task was targeted. See Chapter 56, *The Build Lifecycle* for more details about the build lifecycle.

Example 15.11. Adding dependency on task from another project

build.gradle

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

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

Output of **gradle -q taskX**

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

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

Example 15.12. Adding dependency using task object

build.gradle

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

task taskY << {
    println 'taskY'
}

taskX.dependsOn taskY
```

Output of **gradle -q taskX**

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

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

Example 15.13. Adding dependency using closure

`build.gradle`

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

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

task lib1 << {
    println 'lib1'
}

task lib2 << {
    println 'lib2'
}

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

Output of **`gradle -q taskX`**

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

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

15.5. 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 *order* 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: eg. 'build' never runs before 'clean'.
- Run build validations early in the build: eg. 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: eg. unit tests should run before integration tests.
- A task that aggregates the results of all tasks of a particular type: eg. 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 15.14. Adding a 'must run after' task ordering

`build.gradle`

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

Output of **`gradle -q taskY taskX`**

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

Example 15.15. Adding a 'should run after' task ordering

`build.gradle`

```
task taskX << {
    println 'taskX'
}
task taskY << {
    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 15.16. 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()` and `Task.shouldRunAfter()` methods. These methods accept a task instance, a task name or any other input accepted by `Task.dependsOn()`.

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

`build.gradle`

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

Output of **`gradle -q taskX`**

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

15.6. Adding a description to a task

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

Example 15.18. 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')
}
```

15.7. 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 15.19. Overwriting a task

`build.gradle`

```
task copy(type: Copy)

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

Output of **`gradle -q copy`**

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

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.

15.8. Skipping tasks

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

15.8.1. 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 15.20. Skipping a task using a predicate

build.gradle

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

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

Output of **gradle hello -PskipHello**

```
> gradle hello -PskipHello
:hello SKIPPED

BUILD SUCCESSFUL

Total time: 1 secs
```

15.8.2. 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 15.21. Skipping tasks with StopExecutionException

build.gradle

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

compile.doFirst {
    // Here you would put arbitrary conditions in real life.
    // But this is used in an integration test so we want defined behavior.
    if (true) { throw new StopExecutionException() }
}
task myTask(dependsOn: 'compile') << {
   println 'I am not affected'
}
```

Output of **gradle -q myTask**

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

This feature is helpful if you work with tasks provided by Gradle. It allows you to add *conditional* execution of the built-in actions of such a task. [7]

15.8.3. 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 15.22. Enabling and disabling tasks

`build.gradle`

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

Output of **`gradle disableMe`**

```
> gradle disableMe
:disableMe SKIPPED

BUILD SUCCESSFUL

Total time: 1 secs
```

15.9. Skipping tasks that are up-to-date

If you are using one of the tasks that come with Gradle, such as a task added by the Java plugin, you might have noticed that Gradle will skip tasks that are up-to-date. This behaviour is also available for your tasks, not just for built-in tasks.

15.9.1. Declaring a task's inputs and outputs

Let's have a look at an example. Here our task generates several output files from a source XML file. Let's run it a couple of times.

Example 15.23. A generator task

build.gradle

```
task transform {
    ext.srcFile = file('mountains.xml')
    ext.destDir = new File(buildDir, 'generated')
    doLast {
        println "Transforming source file."
        destDir.mkdirs()
        def mountains = new XmlParser().parse(srcFile)
        mountains.mountain.each { mountain ->
            def name = mountain.name[0].text()
            def height = mountain.height[0].text()
            def destFile = new File(destDir, "${name}.txt")
            destFile.text = "$name -> ${height}\n"
        }
    }
}
```

Output of **gradle transform**

```
> gradle transform
:transform
Transforming source file.
```

Output of **gradle transform**

```
> gradle transform
:transform
Transforming source file.
```

Notice that Gradle executes this task a second time, and does not skip the task even though nothing has changed. Our example task was defined using an action closure. Gradle has no idea what the closure does and cannot automatically figure out whether the task is up-to-date or not. To use Gradle's up-to-date checking, you need to declare the inputs and outputs of the task.

Each task has an `inputs` and `outputs` property, which you use to declare the inputs and outputs of the task. Below, we have changed our example to declare that it takes the source XML file as an input and produces output to a destination directory. Let's run it a couple of times.

Example 15.24. Declaring the inputs and outputs of a task

`build.gradle`

```
task transform {
    ext.srcFile = file('mountains.xml')
    ext.destDir = new File(buildDir, 'generated')
    inputs.file srcFile
    outputs.dir destDir
    doLast {
        println "Transforming source file."
        destDir.mkdirs()
        def mountains = new XmlParser().parse(srcFile)
        mountains.mountain.each { mountain ->
            def name = mountain.name[0].text()
            def height = mountain.height[0].text()
            def destFile = new File(destDir, "${name}.txt")
            destFile.text = "$name -> ${height}\n"
        }
    }
}
```

Output of **`gradle transform`**

```
> gradle transform
:transform
Transforming source file.
```

Output of **`gradle transform`**

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

Now, Gradle knows which files to check to determine whether the task is up-to-date or not.

The task's `inputs` property is of type `TaskInputs`. The task's `outputs` property is of type `TaskOutputs`.

A task with no defined outputs will *never* be considered up-to-date. For scenarios where the outputs of a task are not files, or for more complex scenarios, the `TaskOutputs.upToDateWhen()` method allows you to calculate programmatically if the tasks outputs should be considered up to date.

A task with only outputs defined will be considered up-to-date if those outputs are unchanged since the previous build.

15.9.2. How does it work?

Before a task is executed for the first time, Gradle takes a snapshot of the inputs. This snapshot contains the set 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.

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()`

15.10. 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 15.25. Task rule

`build.gradle`

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

Output of **`gradle -q pingServer1`**

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

The String parameter is used as a description for the rule, 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 15.26. Dependency on rule based tasks

`build.gradle`

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

task groupPing {
    dependsOn pingServer1, pingServer2
}
```

Output of **`gradle -q groupPing`**

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

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.

15.11. 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 15.27. Adding a task finalizer

`build.gradle`

```
task taskX << {
    println 'taskX'
}
task taskY << {
    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 15.28. Task finalizer for a failing task

`build.gradle`

```
task taskX << {
    println 'taskX'
    throw new RuntimeException()
}
task taskY << {
    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()` method. This method accepts a task instance, a task name, or any other input accepted by `Task.dependsOn()`.

15.12. 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. These imports are customizable (see Appendix E, *Existing IDE Support and how to cope without it*).

16
Working With Files

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

16.1. Locating files

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

Example 16.1. 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.xml`.

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.

16.2. 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()` 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 16.1, “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 16.2. 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 16.3. 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 16.4. Implementing a file collection

`build.gradle`

```
task list << {
    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.

16.3. 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()` method. This creates a `FileTree` defined with a base directory, and optionally some Ant-style include and exclude patterns.

Example 16.5. 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 16.6. 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"
}
```

16.4. 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()` and `Project.tarTree()` 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 16.7. 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'))
```

16.5. 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:

Example 16.8. Specifying a set of files

build.gradle

```
// 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 { zipTre
    }
}
```

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 16.9. Specifying a set of files

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() }
}
```

16.6. 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()` method. To specify the destination directory, use the `CopySpec.into()` method.

Example 16.10. 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 16.11. 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 16.12. 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()` 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 15.9, “Skipping tasks that are up-to-date”).

Example 16.13. Copying files using the copy() method without up-to-date check

build.gradle

```
task copyMethod << {
    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 16.14. 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 58, *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.

16.6.1. Renaming files

Example 16.15. 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')
}
```

16.6.2. Filtering files

Example 16.16. 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]"
    }
}
```

A “token” in a source file that both the “expand” and “filter” operations look for, is formatted like “@tokenName@” for a token named “tokenName”.

16.6.3. 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 16.17. Nested copy specs

build.gradle

```
task nestedSpecs(type: Copy) {
    into 'build/explodedWar'
    exclude '**/*staging*'
    from('src/dist') {
        include '**/*.html'
    }
    into('libs') {
        from configurations.runtime
    }
}
```

16.7. 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 16.18. Using the Sync task to copy dependencies

`build.gradle`

```
task libs(type: Sync) {
    from configurations.runtime
    into "$buildDir/libs"
}
```

16.8. 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 16.19. 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.

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.

16.8.1. Archive naming

The format of *`projectName-version.type`* is used for generated archive file names. For example:

Example 16.20. 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.

There are a number of properties which you can set on an archive task. These are listed below in Table 16.1, “Archive tasks - naming properties”. You can, for example, change the name of the archive:

Example 16.21. 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 16.22. 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 16.1. Archive tasks - naming properties

Property name	Type	Default value	Description
`archiveName`	`String`	*`baseName-appendix-version-classifier.extension`* If any of these properties is empty the trailing `-` is not added to the name.	The base name of the generated archive
`archivePath`	`File`	*`destinationDir/archiveName`*	The absolute path of the generated archive.
`destinationDir`	`File`	Depends on the archive type. JARs and WARs go into *`project.buildDir`*`/libraries`. ZIPs and TARs go into *`project.buildDir`*`/distributions`.	The directory to generate the archive into
`baseName`	`String`	*`project.name`*	The base name portion of the archive file name.
`appendix`	`String`	`null`	The appendix portion of the archive file name.
`version`	`String`	*`project.version`*	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.

16.8.2. Sharing content between multiple archives

You can use the `Project.copySpec()` method to share content between archives.

Often you will want to publish an archive, so that it is usable from another project. This process is described in Chapter 52, *Publishing artifacts*

17
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()`. [8]

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.

17.1. 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 17.1. Using an Ant task

`build.gradle`

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

Output of **`gradle hello`**

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

BUILD SUCCESSFUL

Total time: 1 secs
```

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

Example 17.2. Passing nested text to an Ant task

`build.gradle`

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

Output of **`gradle hello`**

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

BUILD SUCCESSFUL

Total time: 1 secs
```

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

Example 17.3. Passing nested elements to an Ant task

`build.gradle`

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

You can access Ant types in the same way that you access tasks, using the name of the type as the method name. The method call returns the Ant data type, which you can then use directly in your build script. In the following example, we create an Ant `path` object, then iterate over the contents of it.

Example 17.4. Using an Ant type

`build.gradle`

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

More information about `AntBuilder` can be found in 'Groovy in Action' 8.4 or at the Groovy Wiki

17.1.1. 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 17.5. Using a custom Ant task

`build.gradle`

```
task check << {
    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 51.4, “How to declare your dependencies”.

Example 17.6. 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 17.7. Using a custom Ant task and dependency management together

`build.gradle`

```
task check << {
    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')
    }
}
```

17.2. 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 17.8. 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 17.9. Task that depends on Ant target

`build.gradle`

```
ant.importBuild 'build.xml'

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

Output of **`gradle intro`**

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

BUILD SUCCESSFUL

Total time: 1 secs
```

Or, you can add behaviour to an Ant target:

Example 17.10. Adding behaviour to an Ant target

`build.gradle`

```
ant.importBuild 'build.xml'

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

Output of **`gradle hello`**

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

BUILD SUCCESSFUL

Total time: 1 secs
```

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

Example 17.11. Ant target that depends on Gradle task

`build.gradle`

```
ant.importBuild 'build.xml'

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

`build.xml`

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

Output of **`gradle hello`**

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

BUILD SUCCESSFUL

Total time: 1 secs
```

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()` method.

Example 17.12. 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.

17.3. 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 17.13. 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 17.14. 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 17.15. 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 17.16. Getting an Ant reference

build.xml

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

build.gradle

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

17.4. API

The Ant integration is provided by `AntBuilder`.

---

[8] 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

18
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 18.1, “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 18.1. 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

18.1. Choosing a log level

You can use the command line switches shown in Table 18.2, “Log level command-line options” to choose different log levels. In Table 18.3, “Stacktrace command-line options” you find the command line switches which affect stacktrace logging.

Table 18.2. 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 18.3. 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 `DEBUG` log level is chosen, truncated stacktraces are always printed.
`-s` or `--stacktrace`	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 *your* code.)
`-S` or `--full-stacktrace`	The full stacktraces are printed out.

18.2. Writing your own log messages

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

Example 18.1. 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 18.2. 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 18.3. 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')
```

18.3. 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 18.4. 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 18.5. 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.

18.4. 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()` 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 18.6. 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 56.6, “Responding to the lifecycle in the build script”.

- `BuildListener`
- `ProjectEvaluationListener`

- `TaskExecutionGraphListener`
- `TaskExecutionListener`
- `TaskActionListener`

19

The Gradle Daemon

19.1. Enter the daemon

The Gradle daemon (sometimes referred as *the build daemon*) aims to improve the startup and execution time of Gradle.

We came up with several use cases where the daemon is very useful. For some workflows, the user invokes Gradle many times to execute a small number of relatively quick tasks. For example:

- When using test driven development, where the unit tests are executed many times.
- When developing a web application, where the application is assembled many times.
- When discovering what a build can do, where `gradle tasks` is executed a number of times.

For these workflows, it is important that the startup cost of invoking Gradle is as small as possible.

In addition, user interfaces can provide some interesting features if the Gradle model can be built relatively quickly. For example, the daemon might be useful for the following scenarios:

- Content assistance in the IDE
- Live visualisation of the build in a GUI
- Tab completion in a CLI

In general, snappy behavior of the build tool is always handy. If you try using the daemon for your local builds, you won't want to go back.

The Tooling API (see Chapter 63, *Embedding Gradle*) uses the daemon all the time, e.g. you cannot officially use the Tooling API without the daemon. This means that whenever you are using the STS Gradle plugin for Eclipse or the Gradle support in Intellij IDEA, you're already using the Gradle Daemon.

In the future, there are plans for more features in the daemon:

- Snappy up-to-date checks: use native file system change notifications (e.g. via jdk7 nio.2) to preemptively perform up-to-date analysis.
- Even faster builds: preemptively evaluate projects, so that the model is ready when the user next invokes Gradle.
- Did we mention faster builds? The daemon can potentially preemptively download dependencies or check for new versions of snapshot dependencies.
- Utilize a pool of reusable processes available for compilation and testing. For example, both the Groovy and Scala compilers have a large startup cost. The build daemon could maintain a process with Groovy and/or Scala already loaded.

- Preemptive execution of certain tasks, for example compilation. Quicker feedback.
- Fast and accurate bash tab completion.
- Periodically garbage collect the Gradle caches.

19.2. Reusing and expiration of daemons

The basic idea is that the Gradle command forks a daemon process, which performs the actual build. Subsequent invocations of the Gradle command will reuse the daemon, avoiding the startup costs. Sometimes we cannot use an existing daemon because it is busy or its Java version or jvm arguments are different. For exact details on when exactly a new daemon process is forked read the dedicated section below. The daemon process automatically expires after 3 hours of idle time.

Here are all situations in which we fork a new daemon process:

- If the daemon process is currently busy running some job, a brand new daemon process will be started.
- We fork a separate daemon process per Java home. So even if there is some idle daemon waiting for build requests but you happen to run a build with a different Java home then a brand new daemon will be forked.
- We fork a separate daemon process if the jvm arguments for the build are sufficiently different. For example we will not fork a new daemon if a some system property has changed. However if the -Xmx memory setting changed or some fundamental immutable system property changed (e.g. file.encoding) then a new daemon will be forked.
- At the moment the daemon is coupled with a particular version of Gradle. This means that even if some daemon is idle but you are running the build with a different version of Gradle, a new daemon will be started. This also has a consequence for the `--stop` command line instruction: this command will only stop daemons that were started with Gradle version that is executing `--stop`.

We plan to improve the functionality of managing and pooling the daemons in the future.

19.3. Usage and troubleshooting

For command line usage, look at the dedicated section in Appendix D, *Gradle Command Line*. If you are tired of using the same command line options again and again, take a look at Section 20.1, “Configuring the build environment via gradle.properties”. This section contains information on how to configure certain behavior of the daemon (including turning on the daemon by default) in a more 'persistent' way.

Some ways of troubleshooting the Gradle daemon:

- If you have a problem with your build, try temporarily disabling the daemon (you can pass the command line switch `--no-daemon`).
- Occasionally, you may want to stop the daemons either via the `--stop` command line option or in a more forceful way.
- There is a daemon log file, which by default is located in the Gradle user home directory.
- You may want to start the daemon in `--foreground` mode to observe how the build is executed.

19.4. Configuring the daemon

Some daemon settings, such as JVM arguments, memory settings or the Java home, can be configured. Please find more information in Section 20.1, “Configuring the build environment via gradle.properties”

20

The Build Environment

20.1. 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.

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 Section 57.1.1.1, “Configuration on demand”.

`org.gradle.parallel`

When configured, Gradle will run in incubating parallel mode.

20.1.1. 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 19, *The Gradle Daemon* for more details.

20.2. Accessing the web via a proxy

Configuring an HTTP 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 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 20.1. 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 20.2. 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 Subversion view. 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.

20.2.1. 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.

21
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 will discuss how to use plugins and the terminology and concepts surrounding plugins.

21.1. 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

21.2. 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.

21.3. Applying plugins

Plugins are said to be *applied*, which is done via the `Project.apply()` method. The application of plugins is *idempotent*. That is, the same plugin can be applied multiple times. If the plugin has previously been applied, any further applications are safe and will have no effect.

21.3.1. Script plugins

Example 21.1. Applying a script plugin

build.gradle

```
apply from: 'other.gradle'
```

Script plugins 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 build.

21.3.2. Binary plugins

Example 21.2. Applying a binary plugin

build.gradle

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

Plugins can be applied using a *plugin id*. The plugin id serves as a unique identifier for a given plugin. Core plugins register a short name that can be used as the plugin id. In the above case, we are using the short name '`java`' to apply the `JavaPlugin`. Community plugins, on the other hand, use a fully qualified form for the plugin id (e.g. `com.github.foo.bar`), although some legacy plugins may still utilize a short, unqualified form.

Rather than using a plugin id, plugins can also be applied by simply specifying the class of the plugin:

Example 21.3. 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 Appendix E, *Existing IDE Support and how to cope without it*). Furthermore, it is not necessary to append `.class` to identify a class literal in Groovy as it is in Java.

21.3.2.1. Locations of binary plugins

A plugin is simply any class that implements the `Plugin` interface. Gradle provides the core plugins as part of its distribution so simply applying the plugin as above is all you need to do. However, non-core binary plugins need to be available to the build classpath before they can be applied. This can be achieved in a number of ways, including:

- Defining the plugin as an inline class declaration inside a build script.
- Defining the plugin as a source file under the buildSrc directory in the project (see Section 60.3, "Build sources in the `buildSrc` project").
- Including the plugin from an external jar defined as a buildscript dependency (see Section 21.4, "Applying

plugins with the buildscript block").

- Including the plugin from the plugin portal using the plugins DSL (see Section 21.5, "Applying plugins with the plugins DSL").

For more on defining your own plugins, see Chapter 59, *Writing Custom Plugins*.

21.4. 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 60.5, "External dependencies for the build script".

Example 21.4. 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"
```

21.5. 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 more succinct and convenient way to declare plugin dependencies. It works with the new Gradle plugin portal to provide easy access to both core and community plugins. The plugins script block configures an instance of `PluginDependenciesSpec`.

To apply a core plugin, the short name can be used:

Example 21.5. 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 21.6. Applying a community plugin

`build.gradle`

```
plugins {
    id "com.jfrog.bintray" version "0.4.1"
}
```

No further configuration is necessary. Specifically, there is no need to configure the buildscript classpath. Gradle will resolve the plugin in the plugin portal, locate it, and make it available to the build.

See `PluginDependenciesSpec` for more information on using the Plugin DSL.

21.5.1. Limitations of the plugins DSL

The new way to add plugins to a project is much more than a more convenient syntax. The new DSL is processed very differently to the old one. The new mechanism 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.

21.5.1.1. 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»
}
```

Where «`plugin version`» and «`plugin id`» must be constant, literal, strings. 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).

21.5.1.2. 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.

21.5.1.3. Cannot be used in conjunction with subprojects {}, allprojects {}, etc

It is not possible to use the familiar pattern of applying a plugin to multiple projects at once using `subprojects {}`, etc at the moment. There is currently no mechanism for applying a plugin to multiple projects at once. At the moment, each project that requires a plugin must declare so in the `plugins {}` block in its buildscript.

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 buildsc
.

21.6. 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.

21.7. 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 59, *Writing Custom Plugins*.

22
Standard Gradle plugins

There are a number of plugins included in the Gradle distribution. These are listed below.

22.1. Language plugins

These plugins add support for various languages which can be compiled for and executed in the JVM.

Table 22.1. 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 7, *Java Quickstart*.
`groovy`	`java`, `groovy-base`		Adds support for building Groovy projects. See also Chapter 9, *Groovy Quickstart*.
`scala`	`java`, `scala-base`		Adds support for building Scala projects.
`antlr`	`java`	-	Adds support for generating parsers using Antlr.

22.2. Incubating language plugins

These plugins add support for various languages:

Table 22.2. 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.

22.3. Integration plugins

These plugins provide some integration with various runtime technologies.

Table 22.3. Integration plugins

Plugin Id	Automatically applies	Works with	Description
`application`	`java`	-	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 10, *Web Application Quickstart*.
`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 10, *Web Application Quickstart*.

22.4. Incubating integration plugins

These plugins provide some integration with various runtime technologies.

Table 22.4. 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.

22.5. Software development plugins

These plugins provide help with your software development process.

Table 22.5. 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 7, *Java Quickstart*.
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 7, *Java Quickstart*.
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.

`sonar`	-	java-base, java, jacoco	Provides integration with the Sonar code quality platform. Superceeded by the `sonar-runner` plugin.

22.6. Incubating software development plugins

These plugins provide help with your software development process.

Table 22.6. 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.
`sonar-runner`	-	java-base, java, jacoco	Provides integration with the Sonar code quality platform. Supersedes the `sonar` plugin.
`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.

22.7. 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 22.7. Base plugins

Plugin Id	Description
base	Adds the standard lifecycle tasks and configures reasonable defaults for the archive tasks: • adds build *`ConfigurationName`* tasks. Those tasks assemble the artifacts belonging to the specified configuration. • adds upload *`ConfigurationName`* 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 `AbstractArchiveTask`). For example, the archive tasks are tasks of types: `Jar`, `Tar`, `Zip`. Specifically, `destinationDir`, `baseName` and `version` 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.

22.8. Third party plugins

You can find a list of external plugins at the Gradle Plugins site.

23
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.

23.1. Usage

To use the Java plugin, include the following in your build script:

Example 23.1. Using the Java plugin

`build.gradle`

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

23.2. 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.

23.3. Tasks

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

Table 23.1. 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 classes directory.
`classes`	The `compileJava` task and the `processResources` task. Some plugins add additional compilation tasks.	`Task`	Assembles the production classes directory.
`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 classes directory.
`testClasses`	`compileTestJava` task and `processTestResources` task. Some plugins add additional test compilation tasks.	`Task`	Assembles the test classes directory.
`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.
`uploadArchives`	The tasks which produce the artifacts in the `archives` configuration, including `jar`.	`Upload`	Uploads artifacts in the `archives` configuration, including the JAR file.
`clean`	-	`Delete`	Deletes the project build directory.

`clean`*`TaskName`*	-	`Delete`	Deletes files created by specified task. `cleanJar` will delete the JAR file created by the `jar` task, and `cleanTest` will delete the test results created by the `test` task.

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

Table 23.2. Java plugin - source set tasks

Task name	Depends on	Type	Description
`compile`*`SourceSet`*`Java`	All tasks which produce the source set's compile classpath.	`JavaCompile`	Compiles the given source set's Java source files using javac.
`process`*`SourceSet`*`Resources`		`Copy`	Copies the given source set's resources into the classes directory.
`sourceSet``Classes`	The `compile`*`SourceSet`*`Java` task and the `process`*`SourceSet`*`Resources` task. Some plugins add additional compilation tasks for the source set.	`Task`	Assembles the given source set's classes directory.

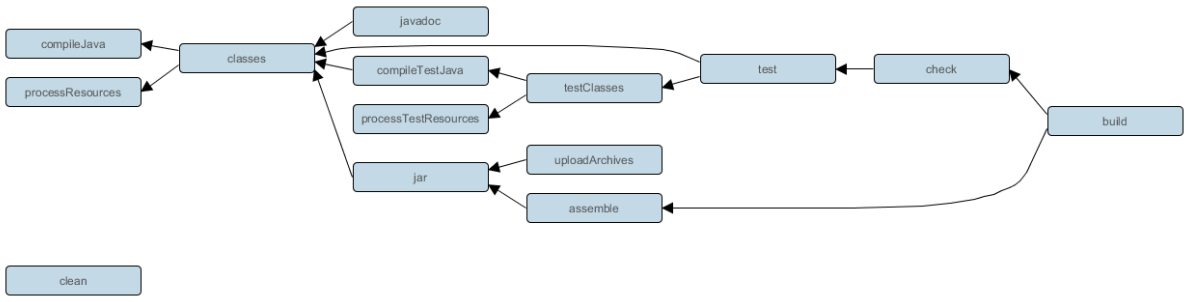
The Java plugin also adds a number of tasks which form a lifecycle for the project:

Table 23.3. Java plugin - lifecycle tasks

Task name	Depends on	Type	Description
`assemble`	All archive tasks in the project, including `jar`. 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 `test`. 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 `testRuntime` 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 `testRuntime` configuration.	`Task`	Performs a full build of the project and all projects which depend on it.
`build`*`ConfigName`*	The tasks which produce the artifacts in configuration *`ConfigName`*.	`Task`	Assembles the artifacts in the specified configuration. The task is added by the Base plugin which is implicitly applied by the Java plugin.
`upload`*`ConfigName`*	The tasks which uploads the artifacts in configuration *`ConfigName`*.	`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 23.1. Java plugin - tasks



23.4. 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 23.4. Java plugin - default project layout

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

23.4.1. 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 23.2. Custom Java source layout

`build.gradle`

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

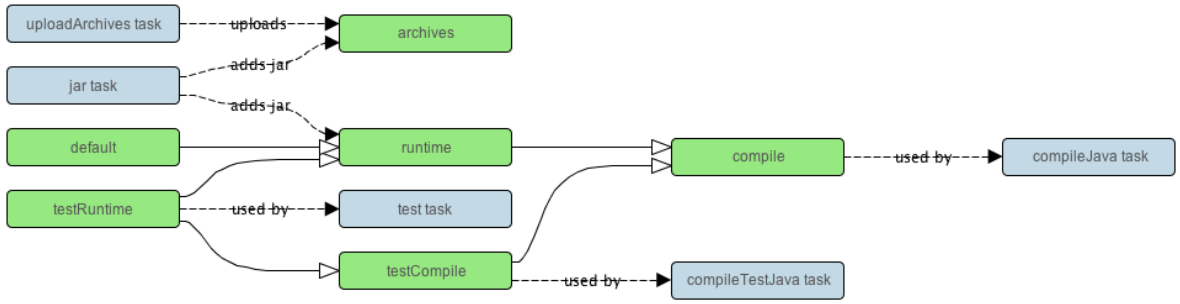
23.5. Dependency management

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 23.5. Java plugin - dependency configurations

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

Figure 23.2. Java plugin - dependency configurations



For each source set you add to the project, the Java plugins adds the following dependency configurations:

Table 23.6. Java plugin - source set dependency configurations

Name	Extends	Used by tasks	Meaning
*sourceSet*Compile		compile*SourceSet*Java	Compile time dependencies for the given source set
*sourceSet*Runtime	*sourceSet*Compile		Runtime dependencies for the given source set

23.6. 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 (see ???).

Table 23.7. Java plugin - directory properties

Property name	Type	Default value	Description

`reportsDirName`	`String`	`reports`	The name of the directory to generate reports into, relative to the build directory.
`reportsDir`	`File` (read-only)	*`buildDir/reportsDirName`*	The directory to generate reports into.
`testResultsDirName`	`String`	`test-results`	The name of the directory to generate test result .xml files into, relative to the build directory.
`testResultsDir`	`File` (read-only)	*`buildDir/testResultsDirName`*	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)	*`reportsDir`*`/testReportDirName`	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)	*`buildDir/libsDirName`*	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)	*`buildDir/distsDirName`*	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)	*`buildDir/docsDirName`*	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)	*`buildDir/dependencyCacheDirName`*	The directory to use to cache source dependency information.

Table 23.8. 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`.	*`sourceCompatibility`*	Java version to generate classes for.
`archivesBaseName`	`String`	*`projectName`*	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`.

23.7. 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 23.3. 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 23.4. Configuring the source directories of a source set

`build.gradle`

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

23.7.1. 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 23.9. 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`	*`buildDir`*`/classes/`*`name`*	The directory to generate the classes of this source set into.
`output.resourcesDir`	`File`	*`buildDir`*`/resources/`*`name`*	The directory to generate the resources of this source set into.
`compileClasspath`	`FileCollection`	`compile`*`SourceSet`* configuration.	The classpath to use when compiling the source files of this source set.
`runtimeClasspath`	`FileCollection`	`output` + `runtime`*`SourceSet`* 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 16.5, “Specifying a set of input files”.	`[`*`projectDir`*`/src/`*`name`*`/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.
`resources.srcDirs`	`Set<File>`. Can set using anything described in Section 16.5, “Specifying a set of input files”.	`[`*`projectDir`*`/src/`*`name`*`/resources]`	The source directories containing the resources of this source set.
`allJava`	`SourceDirectorySet` (read-only)	`java`	All `.java` files of this source set. Some plugins, such as the Groovy plugin, add additional Java source files to this collection.
`allSource`	`SourceDirectorySet` (read-only)	`resources + java`	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.

23.7.2. Defining new source sets

To define a new source set, you simply reference it in the `sourceSets { }` block. Here's an example:

Example 23.5. 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 23.6, “Java plugin - source set dependency configurations”. You can use these configurations to define the compile and runtime dependencies of the source set.

Example 23.6. Defining source set dependencies

`build.gradle`

```
sourceSets {
    intTest
}

dependencies {
    intTestCompile 'junit:junit:4.11'
    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 23.2, “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 23.7. Compiling a source set

Output of **`gradle intTestClasses`**

```
> gradle intTestClasses
:compileIntTestJava
:processIntTestResources
:intTestClasses

BUILD SUCCESSFUL

Total time: 1 secs
```

23.7.3. Some source set examples

Adding a JAR containing the classes of a source set:

Example 23.8. Assembling a JAR for a source set

build.gradle

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

Generating Javadoc for a source set:

Example 23.9. 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 23.10. Running tests in a source set

build.gradle

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

23.8. 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 23.10. Java plugin - Javadoc properties

Task Property	Type	Default Value
`classpath`	`FileCollection`	`sourceSets.main.output` + `sourceSets.m`
`source`	`FileTree`. Can set using anything described in Section 16.5, “Specifying a set of input files”.	`sourceSets.main.allJava`
`destinationDir`	`File`	*`docsDir`*`/javadoc`
`title`	`String`	The name and version of the project

23.9. Clean

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

Table 23.11. Java plugin - Clean properties

Task Property	Type	Default Value
`dir`	`File`	*`buildDir`*

23.10. 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 16.6, “Copying files”.

Table 23.12. Java plugin - ProcessResources properties

Task Property	Type	Default Value
`srcDirs`	`Object`. Can set using anything described in Section 16.5, “Specifying a set of input files”.	*`sourceSet`*`.resources`
`destinationDir`	`File`. Can set using anything described in Section 16.1, “Locating files”.	*`sourceSet`*`.output.resourc`

23.11. 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 23.13. Java plugin - Compile properties

Task Property	Type	Default Value
`classpath`	`FileCollection`	*`sourceSet`*`.compileClass`
`source`	`FileTree`. Can set using anything described in Section 16.5, “Specifying a set of input files”.	*`sourceSet`*`.java`
`destinationDir`	`File`.	*`sourceSet`*`.output.class`

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 `opti` will be honored.

23.12. Incremental Java compilation

Starting with Gradle 2.1, it is possible to compile Java incrementally. This feature is still incubating. 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 everything to avoid problems with constants inlined by the compiler. Down the road we will tune the algorithm and caching so that incremental Java compilation can be a default setting for every compile task.
- 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.

23.13. 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.

23.13.1. 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.

23.13.2. 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).

23.13.3. 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 (`-Dtes` , `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 ceratain 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.
- 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 23.11. 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 all.in.specific.package*`
- `gradle test --tests *IntegTest`
- `gradle test --tests *IntegTest*ui*`
- `gradle someTestTask --tests *UiTest someOtherTestTask --tests *WebTest*ui`

23.13.4. 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`

23.13.5. 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.

23.13.6. 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. [9] The `test` task allows the specification of the JUnit categories you want to include and exclude.

Example 23.12. 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. [10] The test groups that should be included or excluded from the test execution can be configured in the test task.

Example 23.13. Grouping TestNG tests

build.gradle

```
test {
    useTestNG {
        excludeGroups 'integrationTests'
        includeGroups 'unitTests'
    }
}
```

23.13.7. 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 `destinationDi` 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 23.14. 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.

23.13.7.1. 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.

23.13.8. Convention values

Table 23.14. 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`
`testSrcDirs`	`List<File>`	`sourceSets.test.java.srcDirs`

23.14. 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 16.8, “Creating archives” and artifact configurations in Chapter 52, *Publishing artifacts*.

23.14.1. 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 23.15. 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 23.16. 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.

Example 23.17. 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()
                }
            }
        }
    }
}
```

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 `writeTo` or `effectiveManifest`

You can easily write a manifest to disk.

Example 23.18. Separate MANIFEST.MF for a particular archive

build.gradle

```
jar.manifest.writeTo("$buildDir/mymanifest.mf")
```

23.15. Uploading

How to upload your archives is described in Chapter 52, *Publishing artifacts*.

---

[9] The JUnit wiki contains a detailed description on how to work with JUnit categories: https://github.com/junit-team/junit/wiki/Categories.

[10] The TestNG documentation contains more details about test groups: http://testng.org/doc/documentation-main.html#test-groups.

24
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.

24.1. Usage

To use the Groovy plugin, include the following in your build script:

Example 24.1. Using the Groovy plugin

`build.gradle`

```
apply plugin: 'groovy'
```

24.2. Tasks

The Groovy plugin adds the following tasks to the project.

Table 24.1. Groovy plugin - tasks

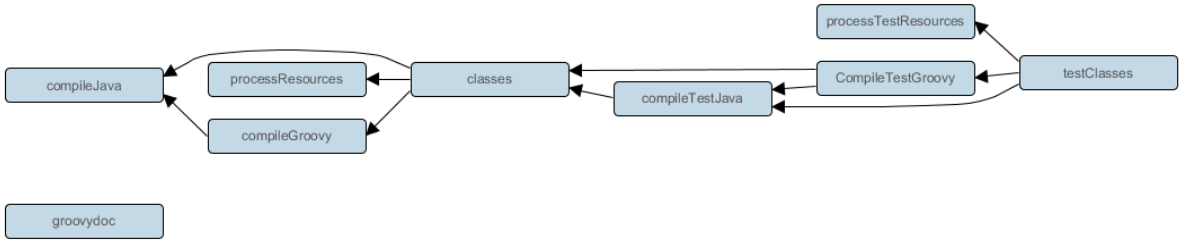
Task name	Depends on	Type	Description
`compileGroovy`	`compileJava`	`GroovyCompile`	Compiles production Groovy source files.
`compileTestGroovy`	`compileTestJava`	`GroovyCompile`	Compiles test Groovy source files.
`compile`*`SourceSet`*`Groovy`	`compile`*`SourceSet`*`Java`	`GroovyCompile`	Compiles the 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 24.2. Groovy plugin - additional task dependencies

Task name	Depends on
classes	compileGroovy
testClasses	compileTestGroovy
*`sourceSet`*Classes	compile*`SourceSet`*Groovy

Figure 24.1. Groovy plugin - tasks



24.3. Project layout

The Groovy plugin assumes the project layout shown in Table 24.3, “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. [11] None of these directories need to exist or have anything in them; the Groovy plugin will simply compile whatever it finds.

Table 24.3. 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.

24.3.1. 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 24.2. Custom Groovy source layout

`build.gradle`

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

    test {
        groovy {
            srcDirs = ['test/groovy']
        }
    }
}
```

24.4. 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 (2.3.3 as of Gradle 2.0). 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 24.3. Configuration of Groovy dependency

`build.gradle`

```
repositories {
    mavenCentral()
}

dependencies {
    compile 'org.codehaus.groovy:groovy-all:2.3.6'
}
```

If Groovy is only used for test code, the Groovy dependency should be added to the `testCompile` configuration:

Example 24.4. Configuration of Groovy test dependency

`build.gradle`

```
dependencies {
    testCompile "org.codehaus.groovy:groovy:2.3.6"
}
```

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 then declaring a regular Groovy dependency.

Example 24.5. 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 24.6. Configuration of Groovy file dependency

`build.gradle`

```
repositories {
    flatDir { dirs 'lib' }
}

dependencies {
    compile module('org.codehaus.groovy:groovy:1.6.0') {
        dependency('asm:asm-all:2.2.3')
        dependency('antlr:antlr:2.7.7')
        dependency('commons-cli:commons-cli:1.2')
        module('org.apache.ant:ant:1.9.3') {
            dependencies('org.apache.ant:ant-junit:1.9.3@jar',
                         'org.apache.ant:ant-launcher:1.9.3')
        }
    }
}
```

The "`module`" reference may be new to you. See Chapter 51, *Dependency Management* for more information about this and other information about dependency management.

24.5. 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.

24.6. Convention properties

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

24.7. 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 (see ???).

Table 24.4. Groovy plugin - source set properties

Property name	Type	Default value	Description
`groovy`	`SourceDirectorySet` (read-only)	Not null	The Groovy source files of this source set. Contains all `.groovy` and `.java` files found in the Groovy source directories, and excludes all other types of files.
`groovy.srcDirs`	`Set<File>`. Can set using anything described in Section 16.5, “Specifying a set of input files”.	`[`*`projectDir`*`/src/`*`name`*`/groovy]`	The source directories containing the Groovy source files of this source set. May also contain Java source files for joint compilation.
`allGroovy`	`FileTree` (read-only)	Not null	All Groovy source files of this source set. Contains only the `.groovy` 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 24.5. Groovy plugin - source set properties

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

24.8. GroovyCompile

The Groovy plugin adds a `GroovyCompile` task for each source set in the project. The task type extends the `Java` task (see Section 23.11, “CompileJava”). The `GroovyCompile` task supports most configuration options of the official Groovy compiler.

Table 24.6. Groovy plugin - GroovyCompile properties

Task Property	Type	Default Value
`classpath`	`FileCollection`	*`sourceSet`*`.compileClasspath`
`source`	`FileTree`. Can set using anything described in Section 16.5, “Specifying a set of input files”.	*`sourceSet`*`.groovy`
`destinationDir`	`File`.	*`sourceSet`*`.output.classesDir`
`groovyClasspath`	`FileCollection`	`groovy` configuration if non-empty; Groovy library found on `classpath` otherwise

---

[11] We are using the same conventions as introduced by Russel Winder's Gant tool (http://gant.codehaus.org).

25
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.

25.1. Usage

To use the Scala plugin, include the following in your build script:

Example 25.1. Using the Scala plugin

`build.gradle`

```
apply plugin: 'scala'
```

25.2. Tasks

The Scala plugin adds the following tasks to the project.

Table 25.1. Scala plugin - tasks

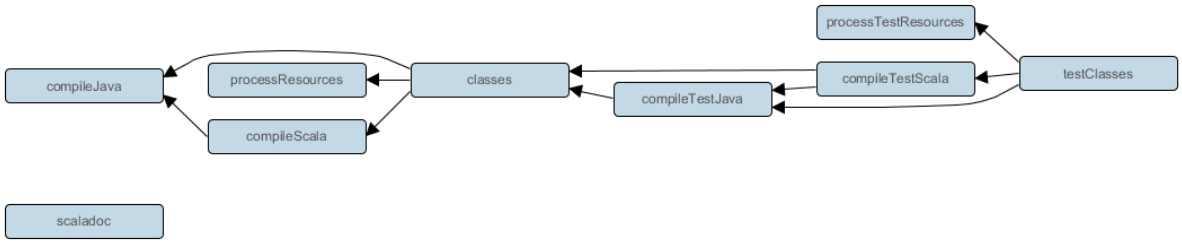
Task name	Depends on	Type	Description
`compileScala`	`compileJava`	`ScalaCompile`	Compiles production Scala source files.
`compileTestScala`	`compileTestJava`	`ScalaCompile`	Compiles test Scala source files.
`compile`*`SourceSet`*`Scala`	`compile`*`SourceSet`*`Java`	`ScalaCompile`	Compiles the given 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 25.2. Scala plugin - additional task dependencies

Task name	Depends on
`classes`	`compileScala`
`testClasses`	`compileTestScala`
`sourceSet``Classes`	`compile`*`SourceSet`*`Scala`

Figure 25.1. Scala plugin - tasks



25.3. 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 25.3. 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.

25.3.1. 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 25.2. Custom Scala source layout

build.gradle

```
sourceSets {
    main {
        scala {
            srcDirs = ['src/scala']
        }
    }
    test {
        scala {
            srcDirs = ['test/scala']
        }
    }
}
```

25.4. 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. [12]

If Scala is used for production code, the `scala-library` dependency should be added to the `compile` configuration:

Example 25.3. Declaring a Scala dependency for production code

build.gradle

```
repositories {
    mavenCentral()
}

dependencies {
    compile 'org.scala-lang:scala-library:2.11.1'
}
```

If Scala is only used for test code, the `scala-library` dependency should be added to the `testCompile` configuration:

Example 25.4. Declaring a Scala dependency for test code

build.gradle

```
dependencies {
    testCompile "org.scala-lang:scala-library:2.11.1"
}
```

25.5. 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.

25.6. Convention properties

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

25.7. Source set properties

The Scala plugin adds the following convention properties to each source set in the project. You can use these properties in your build script as though they were properties of the source set object (see ???).

Table 25.4. 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 16.5, "Specifying a set of input files".	`[projectDir/src/name/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 25.5. 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.

25.8. Fast Scala Compiler

The Scala plugin includes support for fsc, the Fast Scala Compiler. `fsc` runs in a separate daemon process and can speed up compilation significantly.

Example 25.5. Enabling the Fast Scala Compiler

`build.gradle`

```
compileScala {
    scalaCompileOptions.useCompileDaemon = true

    // optionally specify host and port of the daemon:
    scalaCompileOptions.daemonServer = "localhost:4243"
}
```

Note that `fsc` expects to be restarted whenever the *contents* of its compile class path change. (It does detect changes to the compile class path itself.) This makes it less suitable for multi-project builds.

25.9. Compiling in external process

When `scalaCompileOptions.fork` is set to `true`, compilation will take place in an external process. The Ant based compiler (`scalaCompileOptions.useAnt = true`) will fork a new process for every invocation of the `ScalaCompile` task, and does not fork by default. The Zinc based compiler (`scalaCompileO`) will leverage the Gradle compiler daemon, and does so by default.

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 25.6. Adjusting memory settings

`build.gradle`

```
tasks.withType(ScalaCompile) {
    configure(scalaCompileOptions.forkOptions) {
        memoryMaximumSize = '1g'
        jvmArgs = ['-XX:MaxPermSize=512m']
    }
}
```

25.10. 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 now supports incremental compilation by integrating with Zinc, a standalone version of sbt's incremental Scala compiler. To switch the `ScalaCompile` task from the default Ant based compiler to the new Zinc based compiler, set `scalaCompileOptions.useAnt` to `false`:

Example 25.7. Activating the Zinc based compiler

`build.gradle`

```
tasks.withType(ScalaCompile) {
    scalaCompileOptions.useAnt = false
}
```

Except where noted in the API documentation, the Zinc based compiler supports exactly the same configuration options as the Ant based compiler. Note, however, that the Zinc compiler requires Java 6 or higher to run. This means that Gradle itself has to be run with Java 6 or higher.

The Scala plugin adds a configuration named `zinc` to resolve the Zinc library and its dependencies. Gradle will have a default version of the Zinc library, but if you want to override the Zinc version that Gradle uses, add an explicit dependency like `“com.typesafe.zinc:zinc:0.1.4”`. Regardless of which Zinc version is used, Zinc will always use the Scala compiler found on the `scalaTools` configuration.

Just like Gradle's Ant based compiler, the Zinc based 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. With the Zinc based compiler, 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.

Due to the overhead of dependency analysis, a clean compilation or a compilation after a larger code change may take longer than with the Ant based compiler. For CI builds and release builds, we currently recommend to use the Ant based compiler.

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.

25.11. 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.

25.12. 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 facet and a Scala compiler library that matches the Scala version on the project's class path.

---

[12] See Section 25.5, “Automatic configuration of scalaClasspath”.

26
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.

26.1. Usage

To use the War plugin, include the following in your build script:

Example 26.1. Using the War plugin

`build.gradle`

```
apply plugin: 'war'
```

26.2. Tasks

The War plugin adds the following tasks to the project.

Table 26.1. 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 26.2. War plugin - additional task dependencies

Task name	Depends on
assemble	war

Figure 26.1. War plugin - tasks

26.3. Project layout

Table 26.3. War plugin - project layout

Directory	Meaning
`src/main/webapp`	Web application sources

26.4. 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 `com` 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:commons-httpclient:3.0@ja` .

26.5. Convention properties

Table 26.4. War plugin - directory properties

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

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

26.6. 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` [13] configuration are copied to `WEB-INF/lib`.

The `War` class in the API documentation has additional useful information.

26.7. Customizing

Here is an example with the most important customization options:

Example 26.2. 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:4.11"
    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.

---

[13] The `runtime` configuration extends the `compile` configuration.

27
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.

27.1. Usage

To use the Ear plugin, include the following in your build script:

Example 27.1. Using the Ear plugin

`build.gradle`

```
apply plugin: 'ear'
```

27.2. Tasks

The Ear plugin adds the following tasks to the project.

Table 27.1. 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 27.2. Ear plugin - additional task dependencies

Task name	Depends on
assemble	ear

27.3. Project layout

Table 27.3. Ear plugin - project layout

Directory	Meaning
`src/main/application`	Ear resources, such as a META-INF directory

27.4. 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.

27.5. Convention properties

Table 27.4. Ear plugin - directory properties

Property name	Type	Default value	Description
`appDirName`	`String`	`src/main/application`	The name of directory, rel directory.
`libDirName`	`String`	`lib`	The name of the generate
`deploymentDescriptor`	`org.gradle.plugins.ear.descriptor.DeploymentDescriptor`	A deployment descriptor with sensible defaults named `application.xml`	Metadata to the contain . If this file a then the exis be used and configuratio will be ignor

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

27.6. 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.

27.7. Customizing

Here is an example with the most important customization options:

Example 27.2. 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(':war')

    // 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 doe
//      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`.

27.8. 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.

28
The Jetty Plugin

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

28.1. Usage

To use the Jetty plugin, include the following in your build script:

Example 28.1. Using the Jetty plugin

`build.gradle`

```
apply plugin: 'jetty'
```

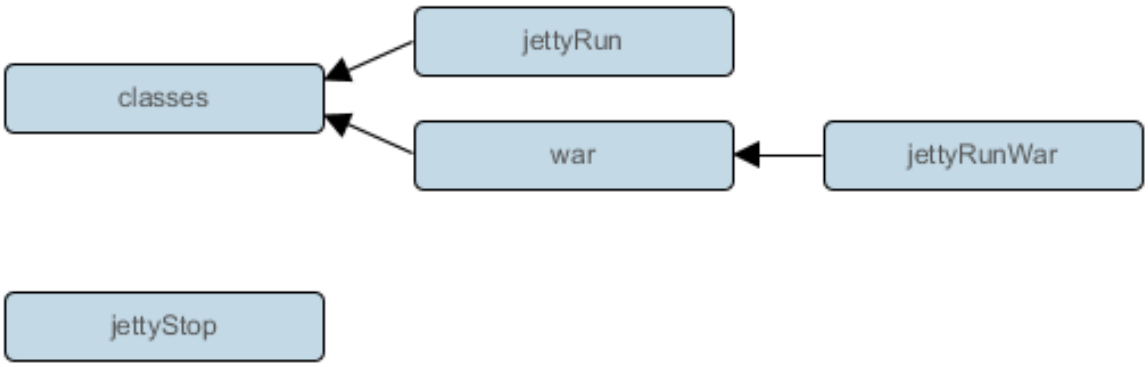
28.2. Tasks

The Jetty plugin defines the following tasks:

Table 28.1. 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 28.1. Jetty plugin - tasks



28.3. Project layout

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

28.4. Dependency management

The Jetty plugin does not define any dependency configurations.

28.5. Convention properties

The Jetty plugin defines the following convention properties:

Table 28.2. 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.

29
The Checkstyle Plugin

The Checkstyle plugin performs quality checks on your project's Java source files using Checkstyle and generates reports from these checks.

29.1. Usage

To use the Checkstyle plugin, include the following in your build script:

Example 29.1. 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`**.

29.2. Tasks

The Checkstyle plugin adds the following tasks to the project:

Table 29.1. 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.
`checkstyle`*`SourceSet`*	*`SourceSet`*`Classes`	`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 29.2. Checkstyle plugin - additional task dependencies

Task name	Depends on
`check`	All Checkstyle tasks, including `checkstyleMain` and `checkstyleTest`.

29.3. Project layout

The Checkstyle plugin expects the following project layout:

Table 29.3. Checkstyle plugin - project layout

File	Meaning
`config/checkstyle/checkstyle.xml`	Checkstyle configuration file

29.4. Dependency management

The Checkstyle plugin adds the following dependency configurations:

Table 29.4. Checkstyle plugin - dependency configurations

Name	Meaning
`checkstyle`	The Checkstyle libraries to use

29.5. Configuration

See the `CheckstyleExtension` class in the API documentation.

30
The CodeNarc Plugin

The CodeNarc plugin performs quality checks on your project's Groovy source files using CodeNarc and generates reports from these checks.

30.1. Usage

To use the CodeNarc plugin, include the following in your build script:

Example 30.1. 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`**.

30.2. Tasks

The CodeNarc plugin adds the following tasks to the project:

Table 30.1. 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.
`codenarc`*`SourceSet`*	-	`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 30.2. CodeNarc plugin - additional task dependencies

Task name	Depends on
`check`	All CodeNarc tasks, including `codenarcMain` and `codenarcTest`.

30.3. Project layout

The CodeNarc plugin expects the following project layout:

Table 30.3. CodeNarc plugin - project layout

File	Meaning
`config/codenarc/codenarc.xml`	CodeNarc configuration file

30.4. Dependency management

The CodeNarc plugin adds the following dependency configurations:

Table 30.4. CodeNarc plugin - dependency configurations

Name	Meaning
`codenarc`	The CodeNarc libraries to use

30.5. Configuration

See the `CodeNarcExtension` class in the API documentation.

31
The FindBugs Plugin

The FindBugs plugin performs quality checks on your project's Java source files using FindBugs and generates reports from these checks.

31.1. Usage

To use the FindBugs plugin, include the following in your build script:

Example 31.1. 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`**.

31.2. Tasks

The FindBugs plugin adds the following tasks to the project:

Table 31.1. 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.
`findbugs`*`SourceSet`*	*`sourceSet`*`Classes`	`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 31.2. FindBugs plugin - additional task dependencies

Task name	Depends on
`check`	All FindBugs tasks, including `findbugsMain` and `findbugsTest`.

31.3. Dependency management

The FindBugs plugin adds the following dependency configurations:

Table 31.3. FindBugs plugin - dependency configurations

Name	Meaning
`findbugs`	The FindBugs libraries to use

31.4. Configuration

See the `FindBugsExtension` class in the API documentation.

32
The JDepend Plugin

The JDepend plugin performs quality checks on your project's source files using JDepend and generates reports from these checks.

32.1. Usage

To use the JDepend plugin, include the following in your build script:

Example 32.1. 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`**.

32.2. Tasks

The JDepend plugin adds the following tasks to the project:

Table 32.1. 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.
`jdepend`*`SourceSet`*	*`sourceSet`*`Classes`	`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 32.2. JDepend plugin - additional task dependencies

Task name	Depends on
`check`	All JDepend tasks, including `jdependMain` and `jdependTest`.

32.3. Dependency management

The JDepend plugin adds the following dependency configurations:

Table 32.3. JDepend plugin - dependency configurations

Name	Meaning
`jdepend`	The JDepend libraries to use

32.4. Configuration

See the `JDependExtension` class in the API documentation.

33
The PMD Plugin

The PMD plugin performs quality checks on your project's Java source files using PMD and generates reports from these checks.

33.1. Usage

To use the PMD plugin, include the following in your build script:

Example 33.1. 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`**.

33.2. Tasks

The PMD plugin adds the following tasks to the project:

Table 33.1. 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.
`pmd`*`SourceSet`*	-	`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 33.2. PMD plugin - additional task dependencies

Task name	Depends on
`check`	All PMD tasks, including `pmdMain` and `pmdTest`.

33.3. Dependency management

The PMD plugin adds the following dependency configurations:

Table 33.3. PMD plugin - dependency configurations

Name	Meaning
`pmd`	The PMD libraries to use

33.4. Configuration

See the `PmdExtension` class in the API documentation.

34
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.

34.1. Getting Started

To get started, apply the JaCoCo plugin to the project you want to calculate code coverage for.

Example 34.1. 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.

34.2. 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 34.2. Configuring JaCoCo plugin settings

`build.gradle`

```
jacoco {
    toolVersion = "0.7.1.201405082137"
    reportsDir = file("$buildDir/customJacocoReportDir")
}
```

Table 34.1. Gradle defaults for JaCoCo properties

Property	Gradle default
reportsDir	“*`$buildDir`*/reports/jacoco”

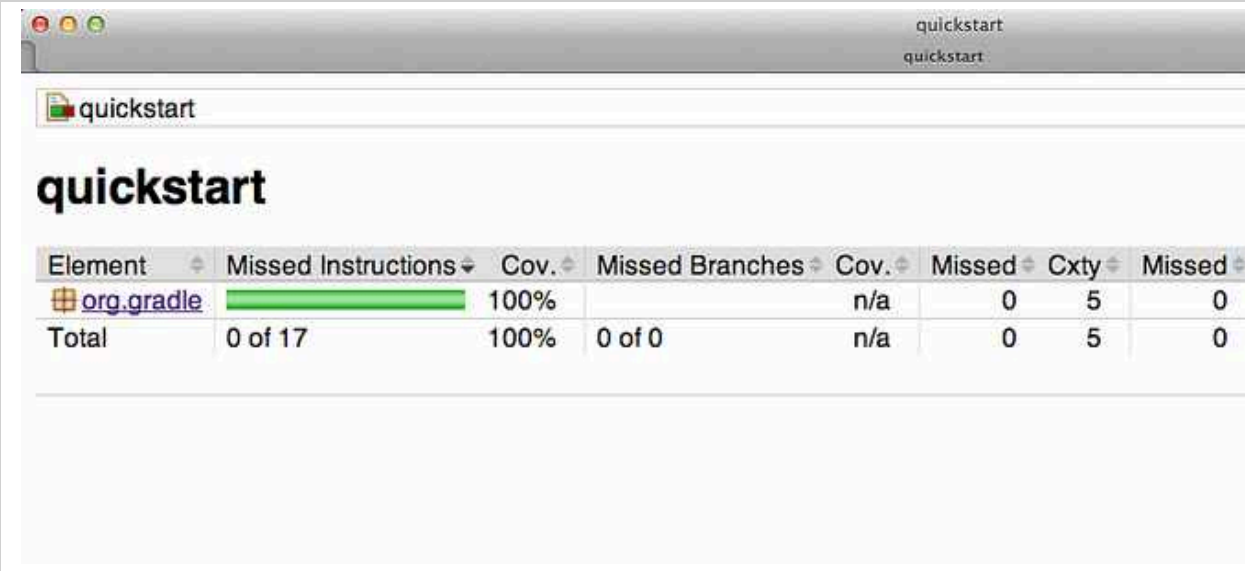
34.3. JaCoCo Report configuration

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 34.3. Configuring test task

`build.gradle`

```
jacocoTestReport {
    reports {
        xml.enabled false
        csv.enabled false
        html.destination "${buildDir}/jacocoHtml"
    }
}
```



34.4. 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 34.4. Configuring test task

`build.gradle`

```
test {
    jacoco {
        append = false
        destinationFile = file("$buildDir/jacoco/jacocoTest.exec")
        classDumpFile = file("$buildDir/jacoco/classpathdumps")
    }
}
```

Table 34.2. Default values of the JaCoCo Task extension

Property	Gradle default
enabled	true
destPath	*`$buildDir`*/jacoco
append	true
includes	[]
excludes	[]
excludeClassLoaders	[]
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 34.5. 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 34.6. Coverage reports generated by applicationCodeCoverageReport

Build layout

```
application/
  build/
    jacoco/
      run.exec
    reports/jacoco/applicationCodeCoverageReport/html/
      index.html
```

34.5. Tasks

For projects that also apply the Java Plugin, The JaCoCo plugin automatically adds the following tasks:

Table 34.3. JaCoCo plugin - tasks

Task name	Depends on	Type	Description
`jacocoTestReport`	-	`JacocoReport`	Generates code coverage report for the test task.

34.6. Dependency management

The JaCoCo plugin adds the following dependency configurations:

Table 34.4. JaCoCo plugin - dependency configurations

Name	Meaning
`jacocoAnt`	The JaCoCo Ant library used for running the `JacocoReport` and `JacocoMerge` tasks.
`jacocoAgent`	The JaCoCo agent library used for instrumenting the code under test.

35
The Sonar Plugin

> You may wish to use the new Sonar Runner Plugin instead of this plugin. In particular, only the Sonar Runner plugin supports Sonar 3.4 and higher.

The Sonar plugin provides integration with Sonar, a web-based platform for monitoring code quality. The plugin adds a `sonarAnalyze` task that analyzes the project to which the plugin is applied, as well as its subprojects. The results are stored in the Sonar database. The plugin is based on the Sonar Runner and requires Sonar 2.11 or higher.

The `sonarAnalyze` task is a standalone task that needs to be executed explicitly and doesn't depend on any other tasks. Apart from source code, the task also analyzes class files and test result files (if available). For best results, it is therefore recommended to run a full build before the analysis. In a typical setup, analysis would be performed once per day on a build server.

35.1. Usage

At a minimum, the Sonar plugin has to be applied to the project.

Example 35.1. Applying the Sonar plugin

`build.gradle`

```
apply plugin: "sonar"
```

Unless Sonar is run locally and with default settings, it is necessary to configure connection settings for the Sonar server and database.

Example 35.2. Configuring Sonar connection settings

build.gradle

```
sonar {
    server {
        url = "http://my.server.com"
    }
    database {
        url = "jdbc:mysql://my.server.com/sonar"
        driverClassName = "com.mysql.jdbc.Driver"
        username = "Fred Flintstone"
        password = "very clever"
    }
}
```

Alternatively, some or all connection settings can be set from the command line (see Section 35.6, “Configuring Sonar Settings from the Command Line”).

Project settings determine how the project is going to be analyzed. The default configuration works well for analyzing standard Java projects and can be customized in many ways.

Example 35.3. Configuring Sonar project settings

build.gradle

```
sonar {
    project {
        coberturaReportPath = file("$buildDir/cobertura.xml")
    }
}
```

The `sonar`, `server`, `database`, and `project` blocks in the examples above configure objects of type `SonarRootModel`, `SonarServer`, `SonarDatabase`, and `SonarProject`, respectively. See their API documentation for further information.

35.2. Analyzing Multi-Project Builds

The Sonar plugin is capable of analyzing a whole project hierarchy at once. This yields a hierarchical view in the Sonar web interface with aggregated metrics and the ability to drill down into subprojects. It is also faster than analyzing each project separately.

To analyze a project hierarchy, the Sonar plugin needs to be applied to the top-most project of the hierarchy. Typically (but not necessarily) this will be the root project. The `sonar` block in that project configures an object of type `SonarRootModel`. It holds all global configuration, most importantly server and database connection settings.

Example 35.4. Global configuration in a multi-project build

`build.gradle`

```
apply plugin: "sonar"

sonar {
    server {
        url = "http://my.server.com"
    }
    database {
        url = "jdbc:mysql://my.server.com/sonar"
        driverClassName = "com.mysql.jdbc.Driver"
        username = "Fred Flintstone"
        password = "very clever"
    }
}
```

Each project in the hierarchy has its own project configuration. Common values can be set from a parent build script.

Example 35.5. Common project configuration in a multi-project build

`build.gradle`

```
subprojects {
    sonar {
        project {
            sourceEncoding = "UTF-8"
        }
    }
}
```

The `sonar` block in a subproject configures an object of type `SonarProjectModel`.

Projects can also be configured individually. For example, setting the `skip` property to `true` prevents a project (and its subprojects) from being analyzed. Skipped projects will not be displayed in the Sonar web interface.

Example 35.6. Individual project configuration in a multi-project build

`build.gradle`

```
project(":project1") {
    sonar {
        project {
            skip = true
        }
    }
}
```

Another typical per-project configuration is the programming language to be analyzed. Note that Sonar can only analyze one language per project.

Example 35.7. Configuring the language to be analyzed

build.gradle

```
project(":project2") {
    sonar {
        project {
            language = "groovy"
        }
    }
}
```

When setting only a single property at a time, the equivalent property syntax is more succinct:

Example 35.8. Using property syntax

build.gradle

```
project(":project2").sonar.project.language = "groovy"
```

35.3. Analyzing Custom Source Sets

By default, the Sonar plugin will analyze the production sources in the `main` source set and the test sources in the `test` source set. This works independent of the project's source directory layout. Additional source sets can be added as needed.

Example 35.9. Analyzing custom source sets

build.gradle

```
sonar.project {
    sourceDirs += sourceSets.custom.allSource.srcDirs
    testDirs += sourceSets.integTest.allSource.srcDirs
}
```

35.4. Analyzing languages other than Java

To analyze code written in a language other than Java, install the corresponding Sonar plugin, and set `sonar.proj` accordingly:

Example 35.10. Analyzing languages other than Java

build.gradle

```
sonar.project {
    language = "grvy" // set language to Groovy
}
```

As of Sonar 3.4, only one language per project can be analyzed. You can, however, set a different language for each project in a multi-project build.

35.5. Setting Custom Sonar Properties

Eventually, most configuration is passed to the Sonar code analyzer in the form of key-value pairs known as Sonar properties. The `SonarProperty` annotations in the API documentation show how properties of the plugin's object model get mapped to the corresponding Sonar properties. The Sonar plugin offers hooks to post-process Sonar properties before they get passed to the code analyzer. The same hooks can be used to add additional properties which aren't covered by the plugin's object model.

For global Sonar properties, use the `withGlobalProperties` hook on `SonarRootModel`:

Example 35.11. Setting custom global properties

`build.gradle`

```
sonar.withGlobalProperties { props ->
    props["some.global.property"] = "some value"
    // non-String values are automatically converted to Strings
    props["other.global.property"] = ["foo", "bar", "baz"]
}
```

For per-project Sonar properties, use the `withProjectProperties` hook on `SonarProject`:

Example 35.12. Setting custom project properties

`build.gradle`

```
sonar.project.withProjectProperties { props ->
    props["some.project.property"] = "some value"
    // non-String values are automatically converted to Strings
    props["other.project.property"] = ["foo", "bar", "baz"]
}
```

A list of available Sonar properties can be found in the Sonar documentation. Note that for most of these properties, the Sonar plugin's object model has an equivalent property, and it isn't necessary to use a `withGlobalProperties` or `withProjectProperties` hook. For configuring a third-party Sonar plugin, consult the plugin's documentation.

35.6. Configuring Sonar Settings from the Command Line

The following properties can alternatively be set from the command line, as task parameters of the `sonarAnalyze` task. A task parameter will override any corresponding value set in the build script.

- `server.url`
- `database.url`
- `database.driverClassName`
- `database.username`
- `database.password`

- `showSql`
- `showSqlResults`
- `verbose`
- `forceAnalysis`

Here is a complete example:

```
gradle sonarAnalyze --server.url=http://sonar.mycompany.com
--database.password=myPassword --verbose
```

If you need to set other properties from the command line, you can use system properties to do so:

Example 35.13. Implementing custom command line properties

`build.gradle`

```
sonar.project {
    language = System.getProperty("sonar.language", "java")
}
```

However, keep in mind that it is usually best to keep configuration in the build script and under source control.

35.7. Tasks

The Sonar plugin adds the following tasks to the project.

Table 35.1. Sonar plugin - tasks

Task name	Depends on	Type	Description
`sonarAnalyze`	-	`SonarAnalyze`	Analyzes a project hierarchy and stores the results in the Sonar database.

36
The Sonar Runner Plugin

> The Sonar Runner plugin is currently incubating. Please be aware that the DSL and other configuration may change in later Gradle versions.
>
> It is intended that this plugin will replace the older Sonar Plugin in a future Gradle version.

The Sonar Runner plugin provides integration with Sonar, a web-based platform for monitoring code quality. It is based on the Sonar Runner, a Sonar client component that analyzes source code and build outputs, and stores all collected information in the Sonar database. Compared to using the standalone Sonar Runner, the Sonar Runner plugin offers the following benefits:

Automatic provisioning of Sonar Runner
: The ability to execute the Sonar Runner via a regular Gradle task makes it available anywhere Gradle is available (developer build, CI server, etc.), without the need to manually download, setup, and maintain a Sonar Runner installation.

Dynamic configuration from Gradle build scripts
: All of Gradle's scripting features can be leveraged to configure Sonar Runner as needed.

Extensive configuration defaults
: Gradle already has much of the information needed for Sonar Runner to successfully analyze a project. By preconfiguring the Sonar Runner based on that information, the need for manual configuration is reduced significantly.

36.1. Sonar Runner version and compatibility

The default version of the Sonar Runner used by the plugin is 2.3, which makes it compatible with Sonar 3.0 and higher. For compatibility with Sonar versions earlier than 3.0, you can configure the use of an earlier Sonar Runner version (see Section 36.4, “Specifying the Sonar Runner version”).

36.2. Getting started

To get started, apply the Sonar Runner plugin to the project to be analyzed.

Example 36.1. Applying the Sonar Runner plugin

build.gradle

```
apply plugin: "sonar-runner"
```

Assuming a local Sonar server with out-of-the-box settings is up and running, no further mandatory configuration is required. Execute **`gradle sonarRunner`** and wait until the build has completed, then open the web page indicated at the bottom of the Sonar Runner output. You should now be able to browse the analysis results.

Before executing the `sonarRunner` task, all tasks producing output to be analysed by Sonar need to be executed. Typically, these are compile tasks, test tasks, and code coverage tasks. To meet these needs, the plugins adds a task dependency from `sonarRunner` on `test` if the `java` plugin is applied. Further task dependencies can be added as needed.

36.3. Configuring the Sonar Runner

The Sonar Runner plugin adds a `SonarRunnerRootExtension` extension to the project and a `SonarRunnerExtension` extension to its subprojects, which allows you to configure the Sonar Runner via key/value pairs known as *Sonar properties*. A typical base line configuration includes connection settings for the Sonar server and database.

Example 36.2. Configuring Sonar connection settings

build.gradle

```
sonarRunner {
    sonarProperties {
        property "sonar.host.url", "http://my.server.com"
        property "sonar.jdbc.url", "jdbc:mysql://my.server.com/sonar"
        property "sonar.jdbc.driverClassName", "com.mysql.jdbc.Driver"
        property "sonar.jdbc.username", "Fred Flintstone"
        property "sonar.jdbc.password", "very clever"
    }
}
```

Alternatively, Sonar properties can be set from the command line. See Section 35.6, “Configuring Sonar Settings from the Command Line” for more information.

For a complete list of standard Sonar properties, consult the Sonar documentation. If you happen to use additional Sonar plugins, consult their documentation.

In addition to set Sonar properties, the `SonarRunnerRootExtension` extension allows the configuration of the Sonar Runner version and the `JavaForkOptions` of the forked Sonar Runner process.

The Sonar Runner plugin leverages information contained in Gradle's object model to provide smart defaults for many of the standard Sonar properties. The defaults are summarized in the tables below. Notice that additional defaults are provided for projects that have the `java-base` or `java` plugin applied. For some properties (notably server and database connection settings), determining a suitable default is left to the Sonar Runner.

Table 36.1. Gradle defaults for standard Sonar properties

Property	Gradle default
sonar.projectKey	“$project.group:$project.name” (for root project of analysed hierarchy; left to Sonar Runner otherwise)
sonar.projectName	project.name
sonar.projectDescription	project.description
sonar.projectVersion	project.version
sonar.projectBaseDir	project.projectDir
sonar.working.directory	“$project.buildDir/sonar”
sonar.dynamicAnalysis	“reuseReports”

Table 36.2. Additional defaults when `java-base` plugin is applied

Property	Gradle default
sonar.java.source	project.sourceCompatibility
sonar.java.target	project.targetCompatibility

Table 36.3. Additional defaults when `java` plugin is applied

Property	Gradle default
sonar.sources	sourceSets.main.allSource.srcDirs (filtered to only include existing directories)
sonar.tests	sourceSets.test.allSource.srcDirs (filtered to only include existing directories)
sonar.binaries	sourceSets.main.runtimeClasspath (filtered to only include directories)
sonar.libraries	sourceSets.main.runtimeClasspath (filtering to only include files; `rt.jar` added if necessary)
sonar.surefire.reportsPath	test.testResultsDir (if the directory exists)
sonar.junit.reportsPath	test.testResultsDir (if the directory exists)

Table 36.4. Additional defaults when `jacoco` plugin is applied

Property	Gradle default
sonar.jacoco.reportPath	jacoco.destinationFile

36.4. Specifying the Sonar Runner version

By default, version 2.3 of the Sonar Runner is used. To specify an alternative version, set the `SonarRunnerRootExtension.getToolVersion()` property of the `sonarRunner` extension of the project the plugin was applied to to the desired version. This will result in the Sonar Runner dependency `org.code` being used as the Sonar Runner.

Example 36.3. Configuring Sonar runner version

`build.gradle`

```
sonarRunner {
    toolVersion = '2.3' // default
}
```

36.5. Analyzing Multi-Project Builds

The Sonar Runner is capable of analyzing whole project hierarchies at once. This yields a hierarchical view in the Sonar web interface, with aggregated metrics and the ability to drill down into subprojects. Analyzing a project hierarchy also takes less time than analyzing each project separately.

To analyze a project hierarchy, apply the Sonar Runner plugin to the root project of the hierarchy. Typically (but not necessarily) this will be the root project of the Gradle build. Information pertaining to the analysis as a whole, like server and database connections settings, have to be configured in the `sonarRunner` block of this project. Any Sonar properties set on the command line also apply to this project.

Example 36.4. Global configuration settings

`build.gradle`

```
sonarRunner {
    sonarProperties {
        property "sonar.host.url", "http://my.server.com"
        property "sonar.jdbc.url", "jdbc:mysql://my.server.com/sonar"
        property "sonar.jdbc.driverClassName", "com.mysql.jdbc.Driver"
        property "sonar.jdbc.username", "Fred Flintstone"
        property "sonar.jdbc.password", "very clever"
    }
}
```

Configuration shared between subprojects can be configured in a `subprojects` block.

Example 36.5. Shared configuration settings

build.gradle

```
subprojects {
    sonarRunner {
        sonarProperties {
            property "sonar.sourceEncoding", "UTF-8"
        }
    }
}
```

Project-specific information is configured in the `sonarRunner` block of the corresponding project.

Example 36.6. Individual configuration settings

build.gradle

```
project(":project1") {
    sonarRunner {
        sonarProperties {
            property "sonar.language", "grvy"
        }
    }
}
```

To skip Sonar analysis for a particular subproject, set `sonarRunner.skipProject` to `true`.

Example 36.7. Skipping analysis of a project

build.gradle

```
project(":project2") {
    sonarRunner {
        skipProject = true
    }
}
```

36.6. Analyzing Custom Source Sets

By default, the Sonar Runner plugin passes on the project's `main` source set as production sources, and the project's `test` source set as test sources. This works regardless of the project's source directory layout. Additional source sets can be added as needed.

Example 36.8. Analyzing custom source sets

build.gradle

```
sonarRunner {
    sonarProperties {
        properties["sonar.sources"] += sourceSets.custom.allSource.srcDirs
        properties["sonar.tests"] += sourceSets.integTest.allSource.srcDirs
    }
}
```

36.7. Analyzing languages other than Java

To analyze code written in a language other than Java, you'll need to set `sonar.project.language` accordingly. However, note that your Sonar server has to have the Sonar plugin that handles that programming language.

Example 36.9. Analyzing languages other than Java

`build.gradle`

```
sonarRunner {
    sonarProperties {
        property "sonar.language", "grvy" // set language to Groovy
    }
}
```

As of Sonar 3.4, only one language per project can be analyzed. It is, however, possible to analyze a different language for each project in a multi-project build.

36.8. More on configuring Sonar properties

Let's take a closer look at the `sonarRunner.sonarProperties {}` block. As we have already seen in the examples, the `property()` method allows you to set new properties or override existing ones. Furthermore, all properties that have been configured up to this point, including all properties preconfigured by Gradle, are available via the `properties` accessor.

Entries in the `properties` map can be read and written with the usual Groovy syntax. To facilitate their manipulation, values still have their "idiomatic" type (`File`, `List`, etc.). After the sonarProperties block has been evaluated, values are converted to Strings as follows: Collection values are (recursively) converted to comma-separated Strings, and all other values are converted by calling their `toString()` method.

Because the `sonarProperties` block is evaluated lazily, properties of Gradle's object model can be safely referenced from within the block, without having to fear that they have not yet been set.

36.9. Setting Sonar Properties from the Command Line

Sonar Properties can also be set from the command line, by setting a system property named exactly like the Sonar property in question. This can be useful when dealing with sensitive information (e.g. credentials), environment information, or for ad-hoc configuration.

```
gradle sonarRunner -Dsonar.host.url=http://sonar.mycompany.com -Dsonar.jdbc.passwo
```

While certainly useful at times, we do recommend to keep the bulk of the configuration in a (versioned) build script, readily available to everyone.

A Sonar property value set via a system property overrides any value set in a build script (for the same property). When analyzing a project hierarchy, values set via system properties apply to the root project of the analyzed hierarchy. Each system property starting with "`"sonar."` will taken into account for the sonar runner setup.

36.10. Controlling the Sonar Runner process

The Sonar Runner is executed in a forked process. This allows fine grained control over memory settings, system properties etc. just for the Sonar Runner process. The `forkOptions` property of the `sonarRunner` extension of the project that applies the `sonar-runner` plugin (Usually the `rootProject` but not necessarily) allows the process configuration to be specified. This property is not available in the `SonarRunnerExtension` extension applied to the subprojects.

Example 36.10. setting custom Sonar Runner fork options

`build.gradle`

```
sonarRunner {
    forkOptions {
        maxHeapSize = '512m'
    }
}
```

For a complete reference about the available options, see `JavaForkOptions`.

36.11. Tasks

The Sonar Runner plugin adds the following tasks to the project.

Table 36.5. Sonar Runner plugin - tasks

Task name	Depends on	Type	Description
`sonarRunner`	-	`SonarRunner`	Analyzes a project hierarchy and stores the results in the Sonar database.

37
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 23.14.1, “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.

37.1. Usage

To use the OSGi plugin, include the following in your build script:

Example 37.1. Using the OSGi plugin

`build.gradle`

```
apply plugin: 'osgi'
```

37.2. Implicitly applied plugins

Applies the Java base plugin.

37.3. Tasks

This plugin does not add any tasks.

37.4. Dependency management

TBD

37.5. Convention object

The OSGi plugin adds the following convention object: `OsgiPluginConvention`

37.5.1. Convention properties

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

37.5.2. Convention methods

The OSGi plugin adds the following methods. For more details, see the API documentation of the convention object.

Table 37.1. OSGi methods

Method	Return Type	Description
osgiManifest()	`OsgiManifest`	Returns an OsgiManifest object.
osgiManifest(Closure cl)	`OsgiManifest`	Returns an OsgiManifest 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 37.2. 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.

38
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). Both external dependencies (including associated source and Javadoc files) and project dependencies are considered.

Since version 1.0-milestone-4 of Gradle, the WTP-generating code was refactored into a separate plugin called `ecl`. So if you are interested in WTP integration then only apply the `eclipse-wtp` plugin. Otherwise applying the `eclipse` plugin is enough. This change was requested by Eclipse users who take advantage of the `war` or `ear` plugins, but who don't use Eclipse WTP. Internally, the `eclipse-wtp` plugin also applies the `eclipse` plugin so you don't need to apply both of those plugins.

What exactly the `eclipse` plugin generates depends on which other plugins are used:

Table 38.1. Eclipse plugin behavior

Plugin	Description
None	Generates minimal `.project` file.
Java	Adds Java configuration to `.project`. Generates `.classpath` and JDT settings file.
Groovy	Adds Groovy configuration to `.project` file.
Scala	Adds Scala support to `.project` and `.classpath` files.
War	Adds web application support to `.project` file.
Ear	Adds ear application support to `.project` file.

However, the `eclipse-wtp` plugin *always* 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.

38.1. Usage

To use either the Eclipse or the Eclipse WTP plugin, include one of the lines in your build script:

Example 38.1. Using the Eclipse plugin

build.gradle

```
apply plugin: 'eclipse'
```

Example 38.2. 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.

38.2. Tasks

The Eclipse plugins add the tasks shown below to a project.

Table 38.2. Eclipse plugin - tasks

Task name	Depends on	Type	Description
`eclipse`	all Eclipse configuration file generation tasks	`Task`	Generates all Eclipse
`cleanEclipse`	all Eclipse configuration file clean tasks	`Delete`	Removes all Eclipse c
`cleanEclipseProject`	-	`Delete`	Removes the `.proj`
`cleanEclipseClasspath`	-	`Delete`	Removes the `.clas`
`cleanEclipseJdt`	-	`Delete`	Removes the `.sett` file.
`eclipseProject`	-	`GenerateEclipseProject`	Generates the `.proj`
`eclipseClasspath`	-	`GenerateEclipseClasspath`	Generates the `.clas`
`eclipseJdt`	-	`GenerateEclipseJdt`	Generates the `.sett` file.

Table 38.3. Eclipse WTP plugin - additional tasks

Task name	Depends on	Type	Description
`cleanEclipseWtpComponent`	-	`Delete`	Removes the `.s`
`cleanEclipseWtpFacet`	-	`Delete`	Removes the `.s` file.
`eclipseWtpComponent`	-	`GenerateEclipseWtpComponent`	Generates the `.s`
`eclipseWtpFacet`	-	`GenerateEclipseWtpFacet`	Generates the `.s` file.

38.3. Configuration

Table 38.4. 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.

38.4. 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.

38.4.1. 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.

38.4.1.1. 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`**".

38.4.2. 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 38.5. Advanced configuration hooks

Model	`beforeMerged { arg -> }` argument type	`whenMerged { arg -> }` argument type	`withXml` argument
`EclipseProject`	`Project`	`Project`	`XmlPro`
`EclipseClasspath`	`Classpath`	`Classpath`	`XmlPro`
`EclipseJdt`	`Jdt`	`Jdt`	-
`EclipseWtpComponent`	`WtpComponent`	`WtpComponent`	`XmlPro`
`EclipseWtpFacet`	`WtpFacet`	`WtpFacet`	`XmlPro`

38.4.2.1. 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 38.3. 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 38.4. Partial Overwrite for Project

`build.gradle`

```
eclipse.project.file.beforeMerged { project ->
    project.natures.clear()
}
```

38.4.2.2. 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 38.5. Export Dependencies

`build.gradle`

```
eclipse.classpath.file {
    whenMerged { classpath ->
        classpath.entries.findAll { entry -> entry.kind == 'lib' }*.exported = fals
    }
}
```

38.4.2.3. Modifying the XML representation

The `withXml`hook 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 38.6. 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'
}
```

39
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 39.1. 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.

39.1. Usage

To use the IDEA plugin, include this in your build script:

Example 39.1. 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.

39.2. Tasks

The IDEA plugin adds the tasks shown below to a project. Notice that the `clean` task does not depend on the `clea` 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 39.2. 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.

39.3. Configuration

Table 39.3. 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

39.4. 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.

39.4.1. 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.

39.4.1.1. 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`**".

39.4.2. 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 39.4. Idea plugin hooks

Model	`beforeMerged { arg -> }` argument type	`whenMerged { arg -> }` argument type	`withXml { a` argument type
`IdeaProject`	`Project`	`Project`	`XmlProvider`
`IdeaModule`	`Module`	`Module`	`XmlProvider`
`IdeaWorkspace`	`Workspace`	`Workspace`	`XmlProvider`

39.4.2.1. 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 39.2. 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 39.3. Partial Rewrite for Project

`build.gradle`

```
idea.project.ipr {
    beforeMerged { project ->
        project.modulePaths.clear()
    }
}
```

39.4.2.2. 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 39.4. Export Dependencies

build.gradle

```
idea.module.iml {
    whenMerged { module ->
        module.dependencies*.exported = true
    }
}
```

39.4.2.3. 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 39.5. Customizing the XML

build.gradle

```
idea.project.ipr {
    withXml { provider ->
        provider.node.component
                .find { it.@name == 'VcsDirectoryMappings' }
                .mapping.@vcs = 'Git'
    }
}
```

39.5. 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.

40
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.

40.1. Usage

To use the ANTLR plugin, include the following in your build script:

Example 40.1. Using the ANTLR plugin

`build.gradle`

```
apply plugin: 'antlr'
```

40.2. Tasks

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

Table 40.1. 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.
`generate`*`SourceSet``GrammarSource`	-	`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 40.2. ANTLR plugin - additional task dependencies

Task name	Depends on
compileJava	generateGrammarSource
compileTestJava	generateTestGrammarSource
compile*SourceSet*Java	generate*SourceSet*GrammarSource

40.3. Project layout

Table 40.3. ANTLR plugin - project layout

Directory	Meaning
`src/main/antlr`	Production ANTLR grammar files.
`src/test/antlr`	Test ANTLR grammar files.
`src/`*`sourceSet`*`/antlr`	ANTLR grammar files for the given source set.

40.4. 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 40.2. Declare ANTLR version

`build.gradle`

```
repositories {
    mavenCentral()
}

dependencies {
    antlr "org.antlr:antlr:3.5.2" // use ANTLR version 3
}
```

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.

40.5. Convention properties

The ANTLR plugin does not add any convention properties.

40.6. Source set properties

The ANTLR plugin adds the following properties to each source set in the project.

Table 40.4. 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` 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 16.5, “Specifying a set of input files”.	`[projectDir/src/name/antlr]`	The source directories containing the ANTLR grammar files of this source set.

40.7. 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.

Example 40.3. setting custom max heap size for ANTLR

`build.gradle`

```
generateGrammarSource {
    maxHeapSize = "64m"
}
```

41
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 11.6, “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.

41.1. Usage

To use the Project report plugin, include the following in your build script:

```
apply plugin: 'project-report'
```

41.2. Tasks

The project report plugin defines the following tasks:

Table 41.1. Project report plugin - tasks

Task name	Depends on	Type	Descri
`dependencyReport`	-	`DependencyReportTask`	Genera the pro depend report.
`htmlDependencyReport`	-	`HtmlDependencyReportTask`	Genera an HT depend and depend insight report the pro or a se projec
`propertyReport`	-	`PropertyReportTask`	Genera the pro proper report.
`taskReport`	-	`TaskReportTask`	Genera the pro task re
`projectReport`	`dependencyReport`, `propertyReport`, `taskReport`, `htmlDependencyReport`	`TaskReport`	Genera all pro reports

41.3. Project layout

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

41.4. Dependency management

The project report plugin does not define any dependency configurations.

41.5. Convention properties

The project report defines the following convention properties:

Table 41.2. 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)	*`buildDir/reportsDirName`*	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)	*`reportsDir/projectReportDirName`*	The directory to generate the project report into.

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

42
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)

42.1. Usage

To use the announce plugin, apply it to your build script:

Example 42.1. 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 42.2. Configure the announce plugin

`build.gradle`

```
announce {
  username = 'myId'
  password = 'myPassword'
}
```

Finally, send announcements with the `announce` method:

Example 42.3. Using the announce plugin

`build.gradle`

```
task helloWorld << {
    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 42.1. 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 `sudo` to install it.
local	Windows, Mac OS X, Ubuntu		Automatically chooses between snarl, growl, and notify-send depending on the current operating system.

42.2. Configuration

See the `AnnouncePluginExtension` class in the API documentation.

43
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.

43.1. Usage

To use the build announcements plugin, include the following in your build script:

Example 43.1. 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 43.2. Using the build announcements plugin from an init script

`init.gradle`

```
rootProject {
    apply plugin: 'build-announcements'
}
```

44
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.

44.1. Usage

To use the distribution plugin, include the following in your build script:

Example 44.1. 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`*`/instal`
".

44.2. Tasks

The Distribution plugin adds the following tasks to the project:

Table 44.1. 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`	-	`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 44.2. Multiple distributions - tasks

Task name	Depends on	Type	Description
`${distribution.name}``DistZip`	-	`Zip`	Creates a ZIP archive of the distribution contents
`${distribution.name}``DistTar`	-	`Tar`	Creates a TAR archive of the distribution contents
`install`*`${distribution.name.capitalize()}`*`Dist`		`Sync`	Assembles the distribution content and installs it on the current machine

Example 44.2. Adding extra distributions

`build.gradle`

```
apply plugin: 'distribution'

version = '1.2'
distributions {
    custom {}
}
```

This will add following tasks to the project:

- customDistZip
- customDistTar
- 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/`".

44.3. 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 44.3. 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/dist/main`" 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.

44.4. 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 44.4. publish main distribution

`build.gradle`

```
apply plugin:'maven'

uploadArchives {
    repositories {
        mavenDeployer {
            repository(url: "file://some/repo")
        }
    }
}
```

45
The Application Plugin

The Gradle application plugin extends the language plugins with common application related tasks. It allows running and bundling applications for the jvm by creating a jvm application `Distribution`.

45.1. Usage

To use the application plugin, include the following in your build script:

Example 45.1. Using the application plugin

`build.gradle`

```
apply plugin:'application'
```

To define the main-class for the application you have to set the `mainClassName` property as shown below

Example 45.2. Configure the application main class

`build.gradle`

```
mainClassName = "org.gradle.sample.Main"
```

Then, you can run the application by running **`gradle run`**. Gradle will take care of building the application classes, along with their runtime dependencies, and starting the application with the correct classpath. You can launch the application in debug mode with **`gradle run --debug-jvm`** (see `JavaExec.setDebug()`).

The plugin can also build a distribution for your application. The `Distribution` will package up the runtime dependencies of the application along with some OS specific start scripts. All files stored in `src/dist` will be added to the root of the distribution. 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.

If your Java application requires a specific set of JVM settings or system properties, you can configure the `applica` property. These JVM arguments are applied to the `run` task and also considered in the generated start scripts of your distribution.

Example 45.3. Configure default JVM settings

build.gradle

```
applicationDefaultJvmArgs = ["-Dgreeting.language=en"]
```

45.2. Tasks

The Application plugin adds the following tasks to the project.

Table 45.1. 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.

45.3. 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.

45.4. Including other resources in the distribution

One of the convention properties added by the plugin is `applicationDistribution` which is a `CopySpec`. This specification is used by the `installDist` and `distZip` tasks as the specification of what is to be included in the distribution. In addition to copying the start scripts to the `bin` dir and necessary jars to `lib` in the distribution, all of the files from the `src/dist` directory are also copied. To include any static files in the distribution, simply arrange them in the `src/dist` directory.

If your project generates files to be included in the distribution, e.g. documentation, you can add these files to the distribution by adding to the `applicationDistribution` copy spec.

Example 45.4. 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!")
    }
}

applicationDistribution.from(createDocs) {
    into "docs"
}
```

By specifying that the distribution should include the task's output files (see Section 15.9.1, “Declaring a task's 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 45.5. Automatically creating files for distribution

Output of **`gradle distZip`**

```
> gradle distZip
:createDocs
:compileJava
:processResources UP-TO-DATE
:classes
:jar
:startScripts
:distZip

BUILD SUCCESSFUL

Total time: 1 secs
```

46

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.

46.1. Usage

To use the Java library distribution plugin, include the following in your build script:

Example 46.1. 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 46.2. 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.

46.2. Tasks

The Java library distribution plugin adds the following tasks to the project.

Table 46.1. Java library distribution plugin - tasks

Task name	Depends on	Type	Description
`distZip`	`jar`	`Zip`	Creates a full distribution ZIP archive including runtime libraries.

46.3. 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 46.3. Include files in the distribution

`build.gradle`

```
distributions {
    main {
        baseName = 'my-name'
        contents {
            from { 'src/dist' }
        }
    }
}
```

47
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 21.3, “Applying 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 48, *Wrapper Plugin*), which means that the Gradle Wrapper will also be installed into the project.

47.1. Tasks

The plugin adds the following tasks to the project:

Table 47.1. Build Init plugin - tasks

Task name	Depends on	Type	Description
`init`	`wrapper`	`InitBuild`	Generates a Gradle project.
`wrapper`	`-`	`Wrapper`	Generates Gradle wrapper files.

47.2. 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.

47.3. 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.

47.3.1. “`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. This 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

47.3.2. “`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
- Uses the “`mavenCentral`” 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

47.3.3. “`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
- Uses the “`mavenCentral`” 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

47.3.4. “`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
- Uses the “`mavenCentral`” 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

47.3.5. “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.

48
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 62, *The Gradle Wrapper*.

48.1. 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.

48.2. Tasks

The wrapper plugin adds the following tasks to the project:

Table 48.1. Wrapper plugin - tasks

Task name	Depends on	Type	Description
`wrapper`	-	`Wrapper`	Generates Gradle wrapper files.

49
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.

49.1. Usage

To use the Build Dashboard plugin, include the following in your build script:

Example 49.1. 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.

49.2. Tasks

The Build Dashboard plugin adds the following task to the project:

Table 49.1. Build Dashboard plugin - tasks

Task name	Depends on	Type	Description
`buildDashboard`	-	`GenerateBuildDashboard`	Generates build dashboard report.

49.3. Project layout

The Build Dashboard plugin does not require any particular project layout.

49.4. Dependency management

The Build Dashboard plugin does not define any dependency configurations.

49.5. Configuration

You can influence the location of build dashboard plugin generation via `ReportingExtension`.

50

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.

50.1. Usage

To use the Java Gradle Plugin Development plugin, include the following in your build script:

Example 50.1. Using the Java Gradle Plugin Development plugin

`build.gradle`

```
apply plugin: 'java-gradle-plugin'
```

Applying the plugin automatically applies the Java plugin and adds the `gradleApi()` dependency to the compile configuration. It also decorates the `jar` task with validations.

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.

Any failed validations will result in a warning message.

51
Dependency Management

51.1. 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.

51.1.1. 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.)

51.1.2. 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.

51.2. 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:

51.2.1. 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 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.

51.2.2. 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 a 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.

51.2.3. 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 51.1.2, “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`
- 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.

51.2.4. 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 51.9.3, "Fine-tuned control over dependency caching").

51.3. 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 23.5, "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 51.1. Definition of a configuration

`build.gradle`

```
configurations {
    compile
}
```

To access a configuration:

Example 51.2. Accessing a configuration

build.gradle

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

To configure a configuration:

Example 51.3. Configuration of a configuration

build.gradle

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

51.4. How to declare your dependencies

There are several different types of dependencies that you can declare:

Table 51.1. 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.

51.4.1. External module dependencies

External module dependencies are the most common dependencies. They refer to a module in an external repository.

Example 51.4. 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
    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-3`) 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.

51.4.1.1. 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.`) 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.

51.4.1.2. 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. [14] 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 51.5. 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.

51.4.1.3. Classifiers

The Maven dependency management has the notion of classifiers. [15] Gradle supports this. To retrieve classified dependencies from a Maven repository you can write:

Example 51.6. 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 51.7. Iterating over a configuration

build.gradle

```
task listJars << {
    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
```

51.4.2. 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 51.8. Client module dependencies - transitive dependencies

build.gradle

```
dependencies {
    runtime module("org.codehaus.groovy:groovy:2.3.6") {
        dependency("commons-cli:commons-cli:1.0") {
            transitive = false
        }
        module(group: 'org.apache.ant', name: 'ant', version: '1.9.3') {
            dependencies "org.apache.ant:ant-launcher:1.9.3@jar",
                         "org.apache.ant:ant-junit:1.9.3"
        }
    }
}
```

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.

51.4.3. 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 51.9. Project dependencies

build.gradle

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

For more information see the API documentation for `ProjectDependency`.

Multi-project builds are discussed in Chapter 57, *Multi-project Builds*.

51.4.4. 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 51.10. 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 51.11. Generated file dependencies

build.gradle

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

task compile << {
    println 'compiling classes'
}

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

Output of **gradle -q list**

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

51.4.5. 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 51.12. Gradle API dependencies

build.gradle

```
dependencies {
    compile gradleApi()
}
```

51.4.6. 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 51.13. Gradle's Groovy dependencies

build.gradle

```
dependencies {
    compile localGroovy()
}
```

51.4.7. Excluding transitive dependencies

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

Example 51.14. 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 51.2.3, “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.

51.4.8. 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 51.6, “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 51.15. Optional attributes of dependencies

build.gradle

```
dependencies {
    runtime ":junit:4.10", ":testng"
    runtime name: 'testng'
}
```

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

Example 51.16. Collections and arrays of dependencies

build.gradle

```
List groovy = ["org.codehaus.groovy:groovy-all:2.3.6@jar",
               "commons-cli:commons-cli:1.0@jar",
               "org.apache.ant:ant:1.9.3@jar"]
List hibernate = ['org.hibernate:hibernate:3.0.5@jar',
                  'somegroup:someorg:1.0@jar']
dependencies {
    runtime groovy, hibernate
}
```

51.4.9. 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 51.17. Dependency configurations

build.gradle

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

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

Example 51.18. Dependency configurations for project

`build.gradle`

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

51.4.10. Dependency reports

You can generate dependency reports from the command line (see Section 11.6.4, “Listing project dependencies”). With the help of the Project report plugin (see Chapter 41, *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.

51.5. Working with dependencies

For the examples below we have the following dependencies setup:

Example 51.19. 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 51.20. Accessing declared dependencies

build.gradle

```
task dependencies << {
    configurations.alllife.dependencies.each { dep -> println dep.name }
    println()
    configurations.alllife.allDependencies.each { dep -> println dep.name }
    println()
    configurations.alllife.allDependencies.findAll { dep -> dep.name != 'orca' }
        .each { 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 51.21. Configuration.files

build.gradle

```
task allFiles << {
    configurations.sealife.files.each { file ->
        println file.name
    }
}
```

Output of **gradle -q allFiles**

```
> gradle -q allFiles
orca-1.0.jar
shark-1.0.jar
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 51.22. Configuration.files with spec

`build.gradle`

```
task files << {
    configurations.sealife.files { dep -> dep.name == 'orca' }.each { file ->
        println file.name
    }
}
```

Output of **`gradle -q files`**

```
> gradle -q files
orca-1.0.jar
seal-2.0.jar
```

The `Configuration.files` method always retrieves all artifacts of the *whole* configuration. It then filters the retrieved files by specified dependencies. As you can see in the example, transitive dependencies are included.

You can also copy a configuration. You can optionally specify that only a subset of dependencies from the original configuration should be copied. The copying methods come in two flavors. The `copy` method copies only the dependencies belonging explicitly to the configuration. The `copyRecursive` method copies all the dependencies, including the dependencies from extended configurations.

Example 51.23. Configuration.copy

`build.gradle`

```
task copy << {
    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 51.24. Configuration.copy vs. Configuration.files

`build.gradle`

```
task copyVsFiles << {
    configurations.sealife.copyRecursive { dep -> dep.name == 'orca' }
        .each { file -> println file.name }
    println()
    configurations.sealife.files { dep -> dep.name == 'orca' }
        .each { file -> println file.name }
}
```

Output of **`gradle -q copyVsFiles`**

```
> gradle -q copyVsFiles
orca-1.0.jar
seal-1.0.jar

orca-1.0.jar
seal-2.0.jar
```

In the example above, `orca` has a dependency on `seal-1.0` whereas `shark` has a dependency on `seal-2.0`. The original configuration has therefore a version conflict which is resolved to the newer `seal-2.0` version. The `files` method therefore returns `seal-2.0` as a transitive dependency of`orca`. The copied configuration only has `orca` as a dependency and therefore there is no version conflict and `seal-1.0` is returned as a transitive dependency.

Once a configuration is resolved it is immutable. Changing its state or the state of one of its dependencies will cause an exception. You can always copy a resolved configuration. The copied configuration is in the unresolved state and can be freshly resolved.

To learn more about the API of the configuration class see the API documentation: `Configuration`.

51.6. 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 51.2. 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.

51.6.1. Supported repository transport protocols

Maven and Ivy repositories support the use of various transport protocols. At the moment the following protocols are supported:

Table 51.3. Repository transport protocols

Type	Authentication schemes
`file`	none
`http`	username/password
`https`	username/password
`sftp`	username/password

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 51.25. 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 51.26. 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'
        }
    }
}
```

51.6.2. Maven central repository

To add the central Maven 2 repository (http://repo1.maven.org/maven2) simply add this to your build script:

Example 51.27. Adding central Maven repository

build.gradle

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

Now Gradle will look for your dependencies in this repository.

Warning: Be aware that the central Maven 2 repository is HTTP only and HTTPS is not supported. If you need a public HTTPS enabled central repository, you can use the JCenter public repository (see Section 51.6.3, “Maven JCenter repository”).

51.6.3. 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 51.28. 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 51.29. Using Bintrays's JCenter with HTTP

build.gradle

```
repositories {
    jcenter {
        url "http://jcenter.bintray.com/"
    }
}
```

51.6.4. Local Maven repository

To use the local Maven cache as a repository you can do:

Example 51.30. 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`.

51.6.5. Maven repositories

For adding a custom Maven repository you can do:

Example 51.31. 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 51.32. 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.

51.6.5.1. 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 51.33. 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.

51.6.6. Flat directory repository

If you want to use a (flat) filesystem directory as a repository, simply type:

Example 51.34. 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. If you only work with flat directory resolvers you don't need to set all attributes of a dependency. See Section 51.4.8, “Optional attributes”

51.6.7. Ivy repositories

51.6.7.1. Defining an Ivy repository with a standard layout

Example 51.35. Ivy repository

build.gradle

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

51.6.7.2. 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 51.36. 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'` and `'ivy'`. See `IvyArtifactRepository.layout()` in the API documentation for details of these named layouts.

51.6.7.3. 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 51.37. 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 51.38. 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]/[artifac
            artifact "company-artifacts/[organisation]/[module]/[revision]/[artifact]
            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/` .

Example 51.39. 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
        }
    }
}
```

51.6.7.4. 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 51.40. Ivy repository

build.gradle

```
repositories {
    ivy {
        url 'http://repo.mycompany.com'
        credentials {
            username 'user'
            password 'password'
        }
    }
}
```

51.6.8. Working with repositories

To access a repository:

Example 51.41. Accessing a repository

build.gradle

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

To configure a repository:

Example 51.42. Configuration of a repository

build.gradle

```
repositories {
    flatDir {
        name 'localRepository'
    }
}
repositories {
    localRepository {
        dirs 'lib'
    }
}
repositories.localRepository {
    dirs 'lib'
}
```

51.6.9. More about Ivy resolvers

Gradle, thanks to Ivy under its hood, is extremely flexible regarding repositories:

- There are many options for the protocol to communicate with the repository (e.g. filesystem, http, ssh, 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: [16]

```
// Maven2 layout (if a repository is marked as Maven2 compatible, the organization
someroot/[organisation]/[module]/[revision]/[module]-[revision].[ext]

// Typical layout for an Ivy repository (the organization is not split into subfol
someroot/[organisation]/[module]/[revision]/[type]s/[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 51.43. Definition of a custom repository

`build.gradle`

```
repositories {
    ivy {
        ivyPattern "$projectDir/repo/[organisation]/[module]-ivy-[revision].xml"
        artifactPattern "$projectDir/repo/[organisation]/[module]-[revision](-[classi
    }
}
```

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.

51.7. 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-m` 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.

51.8. 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.

51.8.1. 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 51.2.3, “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 Section 51.8.2.3, “Blacklisting a particular version with a replacement”.

51.8.2. 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.

51.8.2.1. Modelling releaseable 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 51.44. 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'
        }
    }
}
```

51.8.2.2. 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 51.45. Using a custom versioning scheme

build.gradle

```
configurations.all {
    resolutionStrategy.eachDependency { DependencyResolveDetails details ->
        if (details.requested.version == 'default') {
            def version = findDefaultVersionInCatalog(details.requested.group, deta
            details.useVersion version
        }
    }
}

def findDefaultVersionInCatalog(String group, String name) {
    //some custom logic that resolves the default version into a specific version
    "1.0"
}
```

51.8.2.3. 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 51.46. Blacklisting a version with a replacement

build.gradle

```
configurations.all {
    resolutionStrategy.eachDependency { DependencyResolveDetails details ->
        if (details.requested.group == 'org.software' && details.requested.name == 's
            //prefer different version which contains some necessary fixes
            details.useVersion '1.2.1'
        }
    }
}
```

51.8.2.4. 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 51.47. 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', versi
        }
        if (details.requested.name == 'log4j') {
            //prefer 'log4j-over-slf4j' over 'log4j', with fixed version:
            details.useTarget "org.slf4j:log4j-over-slf4j:1.7.7"
        }
    }
}
```

51.8.2.5. Declaring that a legacy library is 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 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 51.48. 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.

51.8.3. 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 51.49. 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 reposit
repositories.withType(IvyArtifactRepository) {
    resolve.dynamicMode = true
}
```

51.8.4. 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 `someStatus` 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 51.50. 'Latest' version selector

`build.gradle`

```
dependencies {
    config1 "org.sample:client:latest.integration"
    config2 "org.sample:client:latest.release"
}

task listConfigs << {
    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 51.51. 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 51.52. 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 51.53. Ivy component metadata rule

`build.gradle`

```
dependencies {
    config6 "org.sample:lib:latest.rc"
    components {
        withModule("org.sample:lib") { ComponentMetadataDetails details, IvyModuleD
            if (ivyModule.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 51.54. 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"]
    }
}
```

51.8.5. 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()` 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 51.55. Component selection rule

build.gradle

```
configurations {
    rejectConfig {
        resolutionStrategy {
            componentSelection {
                // Accept the highest version matching the requested version that isn
                all { ComponentSelection selection ->
                    if (selection.candidate.group == 'org.sample' && selection.cand
                        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 51.56. 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 51.57. Component selection rule with metadata

`build.gradle`

```
configurations {
    metadataRulesConfig {
        resolutionStrategy {
            componentSelection {
                // Reject any versions with a status of 'experimental'
                all { ComponentSelection selection, ComponentMetadata metadata ->
                    if (selection.candidate.group == 'org.sample' && metadata.statu
                        selection.reject("don't use experimental candidates from 'org
                    }
                }
                // Accept the highest version with either a "release" branch or a sta
                withModule('org.sample:api') { ComponentSelection selection, IvyMod
                    if (descriptor.branch != "release" && metadata.status != 'milesto
                        selection.reject("'org.sample:api' must have testing branch o
                    }
                }
            }
        }
    }
}
```

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 51.58. 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()
            }
        }
    }
}
```

51.9. 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.

51.9.1. Key features of the Gradle dependency cache

51.9.1.1. 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.

51.9.1.2. 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 (seeSection 51.9.1.3, “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.

51.9.1.3. 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.

51.9.1.4. 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 it's 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.

51.9.1.5. 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.

51.9.2. Command line options to override caching

51.9.2.1. 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.

51.9.2.2. 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.

51.9.3. 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 51.59. 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 51.60. Changing module cache control

build.gradle

```
configurations.all {
    resolutionStrategy.cacheChangingModulesFor 4, 'hours'
}
```

For more details, take a look at the API documentation for `ResolutionStrategy`.

51.10. 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, [17] 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 though the JCenter public repository to provide a single resolution address for all popular OSS Java artifacts (see Section 51.6.3, "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.

51.10.1. 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 51.4.8, “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.

---

[14] Gradle supports partial multiproject builds (see Chapter 57, *Multi-project Builds*).

[15] http://books.sonatype.com/mvnref-book/reference/pom-relationships-sect-project-relationships.html

[16] At http://ant.apache.org/ivy/history/latest-milestone/concept.html you can learn more about ivy patterns.

[17] 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.

52
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 65, *Ivy Publishing (new)* and Chapter 66, *Maven Publishing (new)*. Please try them out and give us feedback.

52.1. 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.

52.2. 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 `upload`*`ConfigurationName`* and `build`*`Cor`* . [18] Execution of these tasks will build or upload the artifacts belonging to the respective configuration.

Table 23.5, “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 52.5, “More about project libraries”. As with dependencies, you can declare as many custom configurations as you like and assign artifacts to them.

52.3. Declaring artifacts

52.3.1. Archive task artifacts

You can use an archive task to define an artifact:

Example 52.1. 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.

52.3.2. File artifacts

You can also use a file to define an artifact:

Example 52.2. 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 52.3. 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 52.4. 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: g
}
```

52.4. 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 51.6, “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 52.5. 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 51.6.9, “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 `layou`

parameter. If no `url` parameter is supplied, then Gradle will use the first defined `artifactPattern` for uploading, or the first defined `ivyPattern` for uploading Ivy files, if this is set.

Uploading to a Maven repository is described in Section 53.6, “Interacting with Maven repositories”.

52.5. 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 51.4.9, “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 53.6, “Interacting with Maven repositories”.

---

[18] To be exact, the Base plugin provides those tasks. This plugin is automatically applied if you use the Java plugin.

53
The Maven Plugin

This chapter is a work in progress

The Maven plugin adds support for deploying artifacts to Maven repositories.

53.1. Usage

To use the Maven plugin, include the following in your build script:

Example 53.1. Using the Maven plugin

`build.gradle`

```
apply plugin: 'maven'
```

53.2. Tasks

The Maven plugin defines the following tasks:

Table 53.1. 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 `archives` configuration. This configuration has by default only the default jar as an element. To learn more about installing to the local repository, see: Section 53.6.3, “Installing to the local repository”

53.3. Dependency management

The Maven plugin does not define any dependency configurations.

53.4. Convention properties

The Maven plugin defines the following convention properties:

Table 53.2. Maven plugin - properties

Property name	Type	Default value	Description
`pomDirName`	`String`	`poms`	The path of the directory to write the generated POMs, relative to the build directory.
`pomDir`	`File (read-only)`	*`buildDir/pomDirName`*	The directory where the generated POMs are written to.
`conf2ScopeMappings`	`Conf2ScopeMappingContainer`	`n/a`	Instructions for mapping Gradle configurations to Maven scopes. See Section 53.6.4.2, “Dependency mapping”.

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

53.5. 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 53.2. Creating a stand alone pom.

build.gradle

```
task writeNewPom << {
    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`

53.6. Interacting with Maven repositories

53.6.1. 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.

53.6.2. 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 53.3. 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 53.3, “Protocol jars for Maven deployment” (those libraries have transitive dependencies which have transitive dependencies). [19] For example, to use the ssh protocol you can do:

Example 53.4. 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 53.4, “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 53.3. 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 53.4. 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

53.6.3. 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 `archiv` 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.

53.6.4. 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 53.5. 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.arc` 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 53.5. 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 53.6. 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 53.7. 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 == 'runti
}
```

If you have more than one artifact to publish, things work a little bit differently. See Section 53.6.4.1, “Multiple artifacts per project”.

To customize the settings for the Maven installer (see Section 53.6.3, “Installing to the local repository”), you can do:

Example 53.8. Customization of Maven installer

build.gradle

```
install {
    repositories.mavenInstaller {
        pom.version = '1.0Maven'
        pom.artifactId = 'myName'
    }
}
```

53.6.4.1. 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 53.9. 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.

53.6.4.2. 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 53.10. Accessing a mapping configuration

`build.gradle`

```
task mappings << {
    println conf2ScopeMappings.mappings
}
```

Gradle exclude rules are converted to Maven excludes if possible. Such a conversion is possible if in the Gradle exclude rule the group as well as the module name is specified (as Maven needs both in contrast to Ivy). Per-configuration excludes are also included in the Maven POM, if they are convertible.

[19] It is planned for a future release to provide out-of-the-box support for this

54
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 PGP signatures (which is the signature format required for publication to the Maven Central Repository).

54.1. Usage

To use the Signing plugin, include the following in your build script:

Example 54.1. Using the Signing plugin

`build.gradle`

```
apply plugin: 'signing'
```

54.2. Signatory credentials

In order to create PGP 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 14.2, “Gradle properties and system properties”).

```
signing.keyId=24875D73
signing.password=secret
signing.secretKeyRingFile=/Users/me/.gnupg/secring.gpg
```

If specifying this information 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"
    }
}
```

54.3. 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.

54.3.1. 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 54.2. 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 54.3. Signing a configuration output

Output of **`gradle signArchives`**

```
> gradle signArchives
:compileJava
:processResources
:classes
:jar
:signArchives

BUILD SUCCESSFUL

Total time: 1 secs
```

54.3.2. 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 54.4. 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 54.5. 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.

54.3.3. 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 54.6. 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` property to a closure to defer the evaluation. See `SigningExtension.setRequired()` for more information.

54.4. 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 `uploadArchives` task as normal.

54.5. 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()`) method that can be used in the `beforeDeployment()` block in your upload task configuration.

Example 54.7. Signing a POM for deployment

build.gradle

```
uploadArchives {
    repositories {
        mavenDeployer {
            beforeDeployment { MavenDeployment deployment -> signing.signPom(deploy
        }
    }
}
```

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.

55
Building native binaries

> The Gradle support for building native binaries is currently incubating. Please be aware that the DSL and other configuration may change in later Gradle versions.

The various native binary 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.

55.1. Supported languages

The following source languages are currently supported:

- C
- C++
- Objective-C
- Objective-C++
- Assembly
- Windows resources

55.2. 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 55.14, “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 and later.
Windows	GCC with Cywin 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 Cywin 64	Windows XP and later.
UNIX-like	GCC	
UNIX-like	Clang	

55.3. 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 caveat will be removed in a future Gradle version.

To build native binaries, you will need to have a compatible tool chain installed:

55.3.1. 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.

55.3.2. 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.

55.3.3. 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.

55.4. Component model

To build native binaries using Gradle, your project should define one or more *native component*s. 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.so` and `libh` 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 *variant*. Binary variants are discussed in detail below.

55.5. 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 55.1. 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`).

55.6. 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 55.2. 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.

55.7. 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	Locatio
`NativeExecutableSpec`	`NativeExecutableBinarySpec`	*`${component.name}`*`Exe`	`${pro`
`NativeLibrarySpec`	`SharedLibraryBinarySpec`	*`${component.name}`*`Sha`	`${pro`
`NativeLibrarySpec`	`StaticLibraryBinarySpec`	*`${component.name}`*`Sta`	`${pro`

55.7.1. 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.

55.8. 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 55.3. The components report

Output of **`gradle components`**

```
> gradle components
:components

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

Native library 'hello'
----------------------

Source sets
    C++ source 'hello:cpp'
        src/hello/cpp

Binaries
    Shared library 'hello:sharedLibrary'
        build using task: :helloSharedLibrary
        platform: current
        build type: debug
        flavor: default
        tool chain: Tool chain 'clang' (Clang)
        shared library file: build/binaries/helloSharedLibrary/libhello.dylib
    Static library 'hello:staticLibrary'
        build using task: :helloStaticLibrary
        platform: current
        build type: debug
        flavor: default
        tool chain: Tool chain 'clang' (Clang)
        static library file: build/binaries/helloStaticLibrary/libhello.a

Native executable 'main'
------------------------

Source sets
    C++ source 'main:cpp'
        src/main/cpp

Binaries
    Executable 'main:executable'
        build using task: :mainExecutable
        install using task: :installMainExecutable
        platform: current
        build type: debug
        flavor: default
        tool chain: Tool chain 'clang' (Clang)
        executable file: build/binaries/mainExecutable/main

Note: currently not all plugins register their components, so some components may not

BUILD SUCCESSFUL

Total time: 1 secs
```

55.9. Language support

Presently, Gradle supports building native binaries 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 `LanguageSourceSet`s containing source files, one for each language.

- C
- C++
- Objective-C
- Objective-C++
- Assembly
- Windows resources

55.9.1. C++ sources

C++ language support is provided by means of the `'cpp'` plugin.

Example 55.4. 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}/he`
.

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 55.5. 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).

55.9.2. C sources

C language support is provided by means of the `'c'` plugin.

Example 55.6. 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}/headers`.

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 55.7. 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).

55.9.3. Assembler sources

Assembly language support is provided by means of the `'assembler'` plugin.

Example 55.8. 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`.

55.9.4. Objective-C sources

Objective-C language support is provided by means of the `'objective-c'` plugin.

Example 55.9. 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`.

55.9.5. Objective-C++ sources

Objective-C++ language support is provided by means of the `'objective-cpp'` plugin.

Example 55.10. 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`.

55.10. 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 55.11. Settings that apply to all binaries

build.gradle

```
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 55.12. Settings that apply to all shared libraries

build.gradle

```
// For any shared library binaries built with Visual C++,
// define the DLL_EXPORT macro
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 55.13. 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 55.14. 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"
            }
        }
    }
}
```

55.11. 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 55.15. 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 55.16. 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 55.17. 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`.

55.12. 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.

55.12.1. 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 55.18. 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 `NativeExecutableBinary` for the `execut`
.

Example 55.19. 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 bi
                lib library: 'hello', linkage: 'static'
            }
        }
    }
}
```

55.12.2. Project Dependencies

For a component produced in a different Gradle project, the notation is similar.

Example 55.20. 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'
                    }
                }
            }
        }
    }
}
```

55.13. 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.

55.13.1. 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 55.21. 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 55.22. Configuring debug binaries

build.gradle

```
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.

55.13.2. 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 55.23. 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.

55.13.3. 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 55.24. 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.

55.13.4. 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`, `platform` and `flavor` defined for the project. It is possible to override this on a per-component basis, by specifying the set of `targetBuildTypes`, `targetPlatform` and/or `targetFlavo` .

Example 55.25. 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()` method is used to specify a platform that the `NativeExecutableSpec` named `main` should be built for.

A similar mechanism exists for selecting `TargetedNativeComponent.targetBuildTypes()` and `TargetedNativeComponent.targetFlavors()`.

55.13.5. 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 55.26. Building all possible variants

`build.gradle`

```
task buildAllExecutables {
    dependsOn binaries.withType(NativeExecutableBinary).matching {
        it.buildable
    }
}
```

55.14. 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.

55.14.1. Defining tool chains

The supported tool chain types are:

- `Gcc`
- `Clang`
- `VisualCpp`

Example 55.27. 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).

55.14.2. 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 55.28. 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"
                }
            }
        }
    }
}
```

55.14.3. 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 55.29. 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"
        }
    }
}
```

55.15. 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.

55.16. 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`.

55.16.1. 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 sources. 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.

Example 55.30. 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);
}
```

Due to this mechanism, your CUnit sources may not contain a `main` method since this will clash with the method provided by Gradle.

55.16.2. 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 55.31. Registering CUnit tests

`build.gradle`

```
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`.

55.16.3. 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 55.32. Running CUnit tests

`build.gradle`

```
apply plugin: "c"
apply plugin: "cunit"

model {
    flavors {
        passing
        failing
    }
    platforms {
        x86 {
            architecture "x86"
        }
    }
    repositories {
        libs(PrebuiltLibraries) {
            cunit {
                headers.srcDir "lib/cunit/2.1-2/include"
                binaries.withType(StaticLibraryBinary) {
                    staticLibraryFile =
                        file("lib/cunit/2.1-2/lib/" +
                             findCUnitLibForPlatform(targetPlatform))
                }
            }
        }
    }
    components {
        operators(NativeLibrarySpec) {
            targetPlatform "x86"
        }
    }
}
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 runFailingOperatorsTestCUnitExe`**

```
> gradle -q runFailingOperatorsTestCUnitExe
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.

56
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*.

56.1. 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 Section 57.1.1.1, “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.

56.2. 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 57, *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 60, *Organizing Build Logic*). Let's first do some introspection with a single project build:

Example 56.1. Single project build

`settings.gradle`

```
println 'This is executed during the initialization phase.'
```

`build.gradle`

```
println 'This is executed during the configuration phase.'

task configured {
    println 'This is also executed during the configuration phase.'
}

task test << {
    println 'This is executed during the execution phase.'
}

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.

56.3. 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 57, *Multi-project Builds*).

56.3.1. 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.

56.3.2. 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.

56.3.2.1. Hierarchical layouts

Example 56.2. 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'.

56.3.2.2. Flat layouts

Example 56.3. 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.

56.3.3. Modifying elements of the project tree

The multi-project tree created in the settings file is made up of so called *project descriptors*. You can modify these descriptors in the settings file at any time. To access a descriptor you can do:

Example 56.4. Modification of elements of the project tree

`settings.gradle`

```
println rootProject.name
println project(':projectA').name
```

Using this descriptor you can change the name, project directory and build file of a project.

Example 56.5. Modification of elements of the project tree

`settings.gradle`

```
rootProject.name = 'main'
project(':projectA').projectDir = new File(settingsDir, '../my-project-a')
project(':projectA').buildFileName = 'projectA.gradle'
```

Look at the `ProjectDescriptor` class in the API documentation for more information.

56.4. 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. [20] 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 57, *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 57.4, "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.

56.5. 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 57, *Multi-project Builds*.

56.6. 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.

56.6.1. 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 56.6. Adding of test task to each project which has certain property set

`build.gradle`

```
allprojects {
    afterEvaluate { project ->
        if (project.hasTests) {
            println "Adding test task to $project"
            project.task('test') << {
                println "Running tests for $project"
            }
        }
    }
}
```

`projectA.gradle`

```
hasTests = true
```

Output of **`gradle -q test`**

```
> gradle -q test
Adding test task to project ':projectA'
Running tests for project ':projectA'
```

This example uses method `Project.afterEvaluate()` to add a closure which is executed after the project is evaluated.

It is also possible to receive notifications when any project is evaluated. This example performs some custom logging of project evaluation. Notice that the `afterProject` notification is received regardless of whether the project evaluates successfully or fails with an exception.

Example 56.7. 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.

56.6.2. 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 56.8. 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.

56.6.3. 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 6.13, “Configure by DAG”.

You can also add a `TaskExecutionGraphListener` to the `TaskExecutionGraph` to receive these events.

56.6.4. 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 56.9. Logging of start and end of each task execution

`build.gradle`

```
task ok

task broken(dependsOn: ok) << {
    throw new RuntimeException('broken')
}

gradle.taskGraph.beforeTask { Task 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.

---

[20] Gradle supports partial multiproject builds (see Chapter 57, *Multi-project Builds*).

57
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.

57.1. 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.

57.1.1. Configuration and execution

Section 56.1, “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.

57.1.1.1. 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 57.9, “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 20.1, “Configuring the build environment via gradle.properties”. To configure on demand just for a given build please see Appendix D, *Gradle Command Line*.

57.1.2. 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 57.1. 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/firstExampl`** 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 57.2. Build script of water (parent) project

`build.gradle`

```
Closure cl = { task -> println "I'm $task.project.name" }
task hello << cl
project(':bluewhale') {
    task hello << cl
}
```

Output of **`gradle -q hello`**

```
> gradle -q hello
I'm water
I'm bluewhale
```

Gradle allows you to access any project of the multi-project build from any build script. The Project API provides a method called `project()`, which takes a path as an argument and returns the Project object for this path. The capability to configure a project build from any build script we call *cross project configuration*. Gradle implements this via *configuration injection*.

We are not that happy with the build script of the `water` project. It is inconvenient to add the task explicitly for every project. We can do better. Let's first add another project called `krill` to our multi-project build.

Example 57.3. 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/wa`** 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 57.4. Water project build script

`build.gradle`

```
allprojects {
    task hello << { task -> println "I'm $task.project.name" }
}
```

Output of **`gradle -q hello`**

```
> gradle -q hello
I'm water
I'm bluewhale
I'm krill
```

Is this cool or is this cool? And how does this work? The Project API provides a property `allprojects` which returns a list with the current project and all its subprojects underneath it. If you call `allprojects` with a closure, the statements of the closure are delegated to the projects associated with `allprojects`. You could also do an iteration via `allprojects.each`, but that would be more verbose.

Other build systems use inheritance as the primary means for defining common behavior. We also offer inheritance for projects as you will see later. But Gradle uses configuration injection as the usual way of defining common behavior. We think it provides a very powerful and flexible way of configuring multiproject builds.

Another possibilty for sharing configuration is to use a common external script. See Section 14.3, “Configuring the project using an external build script” for more information.

57.2. Subproject configuration

The Project API also provides a property for accessing the subprojects only.

57.2.1. Defining common behavior

Example 57.5. Defining common behavior of all projects and subprojects

`build.gradle`

```
allprojects {
    task hello << {task -> println "I'm $task.project.name" }
}
subprojects {
    hello << {println "- I depend on water"}
}
```

Output of **`gradle -q hello`**

```
> gradle -q hello
I'm water
I'm bluewhale
- I depend on water
I'm krill
- I depend on water
```

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 it's 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 any number of code blocks providing additional configuration.

57.2.2. 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 57.6. Defining specific behaviour for particular project

`build.gradle`

```
allprojects {
    task hello << {task -> println "I'm $task.project.name" }
}
subprojects {
    hello << {println "- I depend on water"}
}
project(':bluewhale').hello << {
    println "- I'm the largest animal that has ever lived on this planet."
}
```

Output of **`gradle -q hello`**

```
> gradle -q hello
I'm water
I'm bluewhale
- I depend on water
- I'm the largest animal that has ever lived on this planet.
I'm krill
- I depend on water
```

As we have said, we usually prefer to put project specific behavior into the build script of this project. Let's refactor and also add some project specific behavior to the `krill` project.

Example 57.7. 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/spreadSpeci`** 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 << {task -> println "I'm $task.project.name" }
}
subprojects {
    hello << {println "- I depend on water"}
}
```

Output of **`gradle -q hello`**

```
> gradle -q hello
I'm water
I'm bluewhale
- I depend on water
- I'm the largest animal that has ever lived on this planet.
I'm krill
- I depend on water
- The weight of my species in summer is twice as heavy as all human beings.
```

57.2.3. 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.

57.2.3.1. Filtering by name

Example 57.8. 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/addTropical`** in the '-all' distribution of Gradle.

`settings.gradle`

```
include 'bluewhale', 'krill', 'tropicalFish'
```

`build.gradle`

```
allprojects {
    task hello << {task -> println "I'm $task.project.name" }
}
subprojects {
    hello << {println "- I depend on water"}
}
configure(subprojects.findAll {it.name != 'tropicalFish'}) {
    hello << {println '- I love to spend time in the arctic waters.'}
}
```

Output of **`gradle -q hello`**

```
> gradle -q hello
I'm water
I'm bluewhale
- I depend on water
- I love to spend time in the arctic waters.
- I'm the largest animal that has ever lived on this planet.
I'm krill
- I depend on water
- I love to spend time in the arctic waters.
- The weight of my species in summer is twice as heavy as all human beings.
I'm tropicalFish
- I depend on water
```

The `configure()` method takes a list as an argument and applies the configuration to the projects in this list.

57.2.3.2. Filtering by properties

Using the project name for filtering is one option. Using extra project properties is another. (See Section 13.4.2, "Extra properties" for more information on extra properties.)

Example 57.9. Adding custom behaviour to some projects (filtered by project properties)

Build layout

```
water/
  build.gradle
  settings.gradle
  bluewhale/
    build.gradle
  krill/
    build.gradle
  tropicalFish/
    build.gradle
```

> **Note:** The code for this example can be found at **`samples/userguide/multiproject/tropicalWit`** 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 bein
}
```

`tropicalFish/build.gradle`

```
ext.arctic = false
```

`build.gradle`

```
allprojects {
    task hello << {task -> println "I'm $task.project.name" }
}
subprojects {
    hello {
        doLast {println "- I depend on water"}
        afterEvaluate { Project project ->
            if (project.arctic) { doLast {
                println '- I love to spend time in the arctic waters.' }
            }
        }
    }
}
```

Output of **`gradle -q hello`**

```
> gradle -q hello
I'm water
I'm bluewhale
- I depend on water
- I'm the largest animal that has ever lived on this planet.
- I love to spend time in the arctic waters.
I'm krill
- I depend on water
- The weight of my species in summer is twice as heavy as all human beings.
- I love to spend time in the arctic waters.
I'm tropicalFish
- I depend on water
```

In the build file of the `water` project we use an `afterEvaluate` notification. This means that the closure we are passing gets evaluated *after* the build scripts of the subproject are evaluated. As the property `arctic` is set in those build scripts, we have to do it this way. You will find more on this topic in Section 57.6, “Dependencies - Which dependencies?”

57.3. 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 57.10. 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 57.11. Evaluation and execution of projects

`bluewhale/build.gradle`

```
ext.arctic = true
hello << { println "- I'm the largest animal that has ever lived on this planet." }

task distanceToIceberg << {
    println '20 nautical miles'
}
```

`krill/build.gradle`

```
ext.arctic = true
hello << {
    println "- The weight of my species in summer is twice as heavy as all human bein
}

task distanceToIceberg << {
    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 57.12. 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!

57.4. 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 57.5, “Project and task paths”):

Example 57.13. 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.

57.5. 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.

Originally Gradle used the `'/'` character as a natural path separator. With the introduction of directory tasks (see Section 14.1, "Directory creation") this was no longer possible, as the name of the directory task contains the `'/'` character.

57.6. 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.

57.6.1. Execution dependencies

57.6.1.1. Dependencies and execution order

Example 57.14. Dependencies and execution order

Build layout

```
messages/
  settings.gradle
  consumer/
    build.gradle
  producer/
    build.gradle
```

> **Note:** The code for this example can be found at **`samples/userguide/multiproject/dependencie`** in the '-all' distribution of Gradle.

`settings.gradle`

```
include 'consumer', 'producer'
```

`consumer/build.gradle`

```
task action << {
    println("Consuming message: ${rootProject.producerMessage}")
}
```

`producer/build.gradle`

```
task action << {
    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 57.15. Dependencies and execution order

Build layout

```
messages/
  settings.gradle
  aProducer/
    build.gradle
  consumer/
    build.gradle
```

`settings.gradle`

```
include 'consumer', 'aProducer'
```

`aProducer/build.gradle`

```
task action << {
    println "Producing message:"
    rootProject.producerMessage = 'Watch the order of execution.'
}
```

`consumer/build.gradle`

```
task action << {
    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 57.16. 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.

57.6.1.2. Declaring dependencies

Example 57.17. Declaring dependencies

Build layout

```
messages/
  settings.gradle
  consumer/
    build.gradle
  producer/
    build.gradle
```

> **Note:** The code for this example can be found at **`samples/userguide/multiproject/dependencie`** in the '-all' distribution of Gradle.

`settings.gradle`

```
include 'consumer', 'producer'
```

`consumer/build.gradle`

```
task action(dependsOn: ":producer:action") << {
    println("Consuming message: ${rootProject.producerMessage}")
}
```

`producer/build.gradle`

```
task action << {
    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 57.18. 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.

57.6.1.3. 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 57.19. Cross project task dependencies

`consumer/build.gradle`

```
task consume(dependsOn: ':producer:produce') << {
    println("Consuming message: ${rootProject.producerMessage}")
}
```

`producer/build.gradle`

```
task produce << {
    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.
```

57.6.2. 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 57.20. Configuration time dependencies

`consumer/build.gradle`

```
def message = rootProject.producerMessage

task consume << {
    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 57.21. Configuration time dependencies - evaluationDependsOn

`consumer/build.gradle`

```
evaluationDependsOn(':producer')

def message = rootProject.producerMessage

task consume << {
    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 57.22. Configuration time dependencies

`consumer/build.gradle`

```
task consume << {
    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.

57.6.3. 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. [21] For the example we use only one build script and do *cross project configuration*.

Example 57.23. 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/dependencie`** 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(dependsOn: assemble) << {
    File explodedDist = mkdir("$buildDir/explodedDist")
    subprojects.each {project ->
        project.tasks.withType(Jar).each {archiveTask ->
            copy {
                from archiveTask.archivePath
                into explodedDist
            }
        }
    }
}
```

We have an interesting set of dependencies. Obviously the `date` and `hello` 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.

57.7. 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 51.4.3, “Project dependencies”, Gradle offers project lib dependencies for this.

Example 57.24. 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/dependencie`** 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. [22]

Example 57.25. 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:4.11"
    }
}

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. [23] 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 57.26. 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:4.11", 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`".

57.7.1. 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.

57.8. 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()` 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 57.9, “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:

- Section 57.1.1.1, “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 20.1, “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.

57.9. 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 51.4.3, “Project dependencies”) and/or task dependencies (Section 6.5, “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 57.1, “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 `subproje` 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`, `subproje` , 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; prefering cross configuration from the root project.
- Avoid changing the configuration of other projects at execution time.

57.10. 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 57.25, “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 57.27. 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 57.28. 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 57.29. 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 57.30. 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.

57.11. Multi Project and buildSrc

Section 60.3, "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.

57.12. 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.

57.13. 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`, `evaluationDependsOnChildrer` and project lib dependencies. [24] 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.

---

[21] 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.

[22] "`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.

[23] 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.

[24] So we are well in the range of the 7 plus 2 Rule :)

58
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 6, *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 15, *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.

58.1. 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 60, *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.

58.2. Writing a simple task class

To implement a custom task class, you extend `DefaultTask`.

Example 58.1. 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 58.2. 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 58.3. 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
```

58.3. 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 58.4. 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 58.5. 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
    }
}
```

58.3.1. 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 60.5, “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 58.6. 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!'
}
```

58.3.2. 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 58.7. 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)
    }
}
```

58.4. 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 it's inputs and outputs are up to date (see Section 15.9, “Skipping tasks that are up-to-date”). 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*.

58.4.1. 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()` action for processing any input file that is out-of-date, and a `IncrementalTaskInputs.removed()` action that executes for any input file that has been removed since the previous execution.

Example 58.8. 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"
        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.

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.

58.4.2. 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()` action will be executed for any input file that was *added* or *modified*, and the `IncrementalTaskInputs.removed()` 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()` action will be executed for every input file, and the `IncrementalTaskInputs.removed()` 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()`.

58.4.3. 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 58.9. 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 58.10. 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()`:

Example 58.11. Running the incremental task with updated input files

`build.gradle`

```
task updateInputs() << {
    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()`:

Example 58.12. Running the incremental task with an input file removed

`build.gradle`

```
task removeInput() << {
    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()` action, and no input files are reported to the `IncrementalTaskInputs.removed()` action:

Example 58.13. Running the incremental task with an output file removed

`build.gradle`

```
task removeOutput() << {
    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()` action, and no input files are reported to the `IncrementalTaskInputs.removed()` action:

Example 58.14. 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
```

59
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.

59.1. 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 60, *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.

59.2. 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()` 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 59.1. A custom plugin

`build.gradle`

```
apply plugin: GreetingPlugin

class GreetingPlugin implements Plugin<Project> {
    void apply(Project project) {
        project.task('hello') << {
            println "Hello from the GreetingPlugin"
        }
    }
}
```

Output of **`gradle -q hello`**

```
> gradle -q hello
Hello from the GreetingPlugin
```

One thing to note is that 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 `Plugin` 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).

59.3. 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 59.2. 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') << {
            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 59.3. 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') << {
            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.

59.4. 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()` method to resolve values to files as late as possible.

Example 59.4. 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) << {
    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()` 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.

59.5. 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 59.5. 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 59.6. 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 `impleme` property identifies the `Plugin` implementation class.

59.5.1. 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.

59.5.2. 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.

59.5.3. 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 Section 21.4, "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 59.7. 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 21.5, "Applying plugins with the plugins DSL") to apply the plugin:

Example 59.8. Applying a community plugin with the plugins DSL

`build.gradle`

```
plugins {
    id "com.jfrog.bintray" version "0.4.1"
}
```

59.5.4. Writing tests for your plugin

You can use the `ProjectBuilder` class to create `Project` instances to use when you test your plugin implementation.

Example 59.9. 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)
    }
}
```

59.5.5. 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 59.10. Using the Java Gradle Plugin Development plugin

build.gradle

```
apply plugin: 'java-gradle-plugin'
```

59.6. Maintaining multiple domain objects

Gradle provides some utility classes for maintaining collections of objects, which work well with the Gradle build language.

Example 59.11. Managing domain objects

`build.gradle`

```
apply plugin: DocumentationPlugin

books {
    quickStart {
        sourceFile = file('src/docs/quick-start')
    }
    userGuide {

    }
    developerGuide {

    }
}

task books << {
    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/customPlugin
quickStart -> /home/user/gradle/samples/userguide/organizeBuildLogic/customPluginWith
userGuide -> /home/user/gradle/samples/userguide/organizeBuildLogic/customPluginWithD
```

The `Project.container()` 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.co` methods, it MUST expose a property named "`name`" as the unique, and constant, name for the object. The `projec` 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.contain` method variants that allow custom instantiation strategies.

60
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. [25] 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 sources` 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.

60.1. 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 60.1. 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 << {
    // 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
```

60.2. Injected configuration

You can use the configuration injection technique discussed in Section 57.1, “Cross project configuration” and Section 57.2, “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 60.2. 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 << {
        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
```

60.3. 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 60.1. 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 23.4, “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 60.3. Custom buildSrc build script

`buildSrc/build.gradle`

```
repositories {
    mavenCentral()
}

dependencies {
    testCompile 'junit:junit:4.11'
}
```

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 60.4. 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.

60.4. 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 60.5. Running another build from a build

`build.gradle`

```
task build(type: GradleBuild) {
    buildFile = 'other.gradle'
    tasks = ['hello']
}
```

`other.gradle`

```
task hello << {
    println "hello from the other build."
}
```

Output of **`gradle -q build`**

```
> gradle -q build
hello from the other build.
```

60.5. 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 60.6. 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 51.4, “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 60.7. A build script with external dependencies

`build.gradle`

```
import org.apache.commons.codec.binary.Base64

buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath group: 'commons-codec', name: 'commons-codec', version: '1.2'
    }
}

task encode << {
    def byte[] encodedString = new Base64().encode('hello world\n'.getBytes())
    println new String(encodedString)
}
```

Output of **`gradle -q encode`**

```
> gradle -q encode
aGVsbG8gd29ybGQK
```

For multi-project builds, the dependencies declared in the a project's build script, are available to the build scripts of all sub-projects.

60.6. 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. [26]

Example 60.8. Ant optional dependencies

`build.gradle`

```
configurations {
    ftpAntTask
}

dependencies {
    ftpAntTask("org.apache.ant:ant-commons-net:1.9.3") {
        module("commons-net:commons-net:1.4.1") {
            dependencies "oro:oro:2.0.8:jar"
        }
    }
}

task ftp << {
    ant {
        taskdef(name: 'ftp',
                classname: 'org.apache.tools.ant.taskdefs.optional.net.FTP',
                classpath: configurations.ftpAntTask.asPath)
        ftp(server: "ftp.apache.org", userid: "anonymous", password: "me@myorg.com"
            fileset(dir: "htdocs/manual")
        }
    }
}
```

This is also 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.

60.7. 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.

---

[25] Which might range from a single class to something very complex.

[26] 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.

61
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 47, *Build Init Plugin*).

61.1. 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 60.3, "Build sources in the `buildSrc` project" for details of this feature).

61.2. 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.

61.3. 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.

61.3.1. 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 61.1. Using init script to perform extra configuration before projects are evaluated

`build.gradle`

```
repositories {
    mavenCentral()
}

task showRepos << {
    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]
```

61.4. External dependencies for the init script

In Section 60.5, “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 61.2. 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 51.4, “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 61.3. 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
```

61.5. Init script plugins

Similar to a Gradle build script or a Gradle settings file, plugins can be applied on init scripts.

Example 61.4. 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. Onl
                        remove repo
                    }
                }

                // add the enterprise repository
                maven {
                    name "STANDARD_ENTERPRISE_REPO"
                    url ENTERPRISE_REPOSITORY_URL
                }
            }
        }
    }
}
```

`build.gradle`

```
repositories{
    mavenCentral()
}

 task showRepositories << {
    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()` 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 61.4, “External dependencies for the init script”

62
The Gradle Wrapper

The Gradle Wrapper (henceforth referred to as the “wrapper”) is the preferred way of starting a Gradle build. The wrapper is a batch script on Windows, and a shell script for other operating systems. When you start a Gradle build via the wrapper, Gradle will be automatically downloaded and used to run the build.

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 adding and configuring a `Wrapper` task in your build script, and then executing it.

Example 62.1. 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 62.2. 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.

62.1. 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.

62.2. 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`".

63
Embedding Gradle

63.1. Introduction to the Tooling API

The 1.0 milestone 3 release brought a new API called the tooling API, which you can use for embedding Gradle into your own custom 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 will be IDEs, CI servers, other UI authors, or integration testing of your Gradle plugins. However, it is open for anyone who needs to embed Gradle in their application.

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 different target versions of Gradle. The tooling API is Gradle wrapper aware and, by default, uses the same target Gradle version as that used by the wrapper-powered project.

Some features that the tooling API provides today:

- You can query Gradle for 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.
- You can execute a build and listen to stdout and stderr logging and progress (e.g. the stuff shown in the 'status bar' when you run on the command line).
- Tooling API can download and install the appropriate Gradle version, similar to the wrapper. Bear in mind that the tooling API is wrapper aware so you should not need to configure a Gradle distribution directly.
- 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 bundle in your application.

In the future we may support other interesting features:

- Performance. The API gives us the opportunity to do lots of caching, static analysis and preemptive work, to make things faster for the user.
- Better progress monitoring and build cancellation. For example, allowing test execution to be monitored.
- Notifications when things in the build change, so that UIs and models can be updated. For example, your Eclipse or IDEA project will update immediately, in the background.
- Validating and prompting for user supplied configuration.
- Prompting for and managing user credentials.

63.2. Tooling API and the Gradle Build Daemon

Please take a look at Chapter 19, *The Gradle Daemon*. The Tooling API uses the daemon all the time. In fact, you cannot officially use the Tooling API without the daemon. This means that subsequent calls to the Tooling API, be it model building requests or task executing requests can be executed in the same long-living process. Chapter 19, *The Gradle Daemon* contains more details about the daemon, specifically information on situations when new daemons are forked.

63.3. Quickstart

As the tooling API is an interface for developers, the Javadoc is the main documentation for it. This is exactly our intention - we don't expect this chapter to grow very much. Instead we will add more code samples and improve the Javadoc documentation. The main entry point to the tooling API is the `GradleConnector`. You can navigate from there to find code samples and other instructions. Another very important set of resources are the ***samples*** that live in "`$gradleHome/samples/toolingApi`". These samples also specify all of the required dependencies for the Tooling API, along with the suggested repositories to obtain the jars from.

64
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.

64.1. 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 invokable “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.

64.2. Current Capabilities

As this is an incubating feature, a limited set of the eventual functionality has been implemented at this time.

64.2.1. 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)

64.2.2. 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.)

64.3. 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.

64.3.1. 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.

64.3.2. 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
}
```

64.3.3. 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 52, *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.

65
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 52, *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 51, *Dependency Management*) and other tools that understand the Ivy format.

65.1. 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 65.1. 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 65.2, "Publications").
- Establishes a rule to automatically create a `PublishToIvyRepository` task for the combination of

each `IvyPublication` added (see Section 65.2, “Publications”), with each `IvyArtifactRepository` added (see Section 65.3, “Repositories”).

65.2. Publications

If you are not familiar with project artifacts and configurations, you should read Chapter 52, *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.

65.2.1. 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 65.1. 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 65.2. Publishing a Java module to Ivy

`build.gradle`

```
publications {
        ivyJava(IvyPublication) {
            from components.java
        }
    }
```

65.2.2. 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 65.3. 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.

65.2.3. 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 `descr` property (see `IvyModuleDescriptorSpec`). The `descriptor` property can also be used to add additional custom elements as children of the `<info>` element.

Example 65.4. 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.

65.2.4. 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 65.5. Customizing the module descriptor file

build.gradle

```
publications {
        ivyCustom(IvyPublication) {
            descriptor.withXml {
                asNode().info[0].appendNode('description',
                                            'A demonstration of ivy descriptor custom
            }
        }
    }
```

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()` 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.

65.2.5. 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 65.6. 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.

65.3. Repositories

Publications are published to repositories. The repositories to publish to are defined by the `PublishingExtension.getRepositories()` container.

Example 65.7. 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.

65.4. 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 65.8. 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 UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:jar
:publishIvyJavaPublicationToIvyRepository

BUILD SUCCESSFUL

Total time: 1 secs
```

65.4.1. 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 65.9. Publishing all publications via the “publish” lifecycle task

Output of **gradle publish**

```
> gradle publish
:generateDescriptorFileForIvyJavaPublication
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:jar
:publishIvyJavaPublicationToIvyRepository
:publish

BUILD SUCCESSFUL

Total time: 1 secs
```

65.5. 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 65.10. 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 Gener 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.

65.6. 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 65.11. 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:4.11', project(':project2')
    }
}

project(":project2") {
    description = "The second project"

    dependencies {
       compile 'commons-collections:commons-collections:3.1'
    }
}

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 65.12. 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="default" visibility="public" extends="runtime"/>
    <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="so
  </publications>
  <dependencies>
    <dependency org="junit" name="junit" rev="4.11" conf="runtime-&gt;default"/>
    <dependency org="org.gradle.sample" name="project2" rev="1.0" conf="runtime-&gt;d
  </dependencies>
</ivy-module>
```

Note that «PUBLICATION-TIME-STA in this example Ivy module descriptor will be the timestamp of when the descriptor was generated.

65.7. 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

66
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 52, *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 51, *Dependency Management*) and other tools that understand the Maven repository format.

66.1. 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 66.1. 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 66.2, "Publications").
- Establishes a rule to automatically create a `PublishToMavenRepository` task for the combination of each `MavenPublication` added (see Section 66.2, "Publications"), with each `MavenArtifactRepository` added (see Section 66.3, "Repositories").
- Establishes a rule to automatically create a `PublishToMavenLocal` task for each `MavenPublication` added (seeSection 66.2, "Publications").

66.2. Publications

If you are not familiar with project artifacts and configurations, you should read the Chapter 52, *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.

66.2.1. 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 66.1. Software Components

Name	Provided By	Artifacts	Dependencies
`java`	Chapter 23, *The Java Plugin*	Generated jar file	Dependencies from 'runtime' configuration
`web`	Chapter 26, *The 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 66.2. Adding a MavenPublication for a Java component

`build.gradle`

```
publishing {
    publications {
        mavenJava(MavenPublication) {
            from components.java
        }
    }
}
```

66.2.2. 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 66.3. 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.

66.2.3. 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 66.4. 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.

66.2.4. 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 66.5. 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()` in the API documentation for more information.

It is possible to modify virtually any aspect of the created POM should you need to. 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.

66.2.5. 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 66.6. 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.

66.3. Repositories

Publications are published to repositories. The repositories to publish to are defined by the `PublishingExtension.getRepositories()` container.

Example 66.7. 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.

66.4. Performing a publish

The "`maven-publish`" plugin automatically creates a `PublishToMavenRepository` task for each `MavenPublication` and `MavenArtifactRepository` combination in the `publishing.publication` and `publishing.repositories` containers respectively.

The created task is named "`publish«`*`PUBNAME`*`»PublicationTo«`*`REPONAME`*`»Repository`".

Example 66.8. 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 UP-TO-DATE
: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.

66.5. 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 66.9. Publish a project to the Maven local repository

Output of **`gradle publishToMavenLocal`**

```
> gradle publishToMavenLocal
:generatePomFileForMavenJavaPublication
:compileJava
:processResources UP-TO-DATE
: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.

66.6. 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 "`generatePomFileForMavenCust`".

Example 66.10. 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 `arti`, 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 `GenerateM` 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.

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/customizedLayout	Groovy project with a custom source layout
groovy/mixedJavaAndGroovy	Project containing a mix of Java and Groovy source
groovy/multiproject	Build made up of multiple Groovy projects. Also demonstrates how to exclude certain source files, and the use of a custom Groovy AST transformation.

`groovy/quickstart`	Groovy quickstart sample
`java/base`	Java base project
`java/customizedLayout`	Java project with a custom source layout
`java/multiproject`	This sample demonstrates how an application can be composed using multiple Java projects.
`java/quickstart`	Java quickstart project
`java/withIntegrationTests`	This sample demonstrates how to use a source set to add an integration test suite to a Java project.
`javaGradlePlugin`	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 gradleApi() dependency. Furthermore, validations are performed against the plugin metadata during jar execution.
`maven/pomGeneration`	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
`scala/customizedLayout`	Scala project with a custom source layout
`scala/fsc`	Scala project using the Fast Scala Compiler (fsc).
`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
`userguide/javaLibraryDistribution`	A project which uses the Java library distribution plugin
`webApplication/customized`	Web application with customized WAR contents.
`webApplication/quickstart`	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.`). 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.

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. [27]

B.2. Configuration and execution phase

It is important to keep in mind that Gradle has a distinct configuration and execution phase (see Chapter 56, *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') << {
    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.

---

[27] One of those discussions can be found here: http://groovy.329449.n5.nabble.com/script-scoping-question-td355887.html

C
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 11.6.2, “Listing tasks”.

`-b`, `--build-file`
Specifies the build file. See Section 11.5, “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 Section 57.1.1.1, “Configuration on demand”.

`-D`, `--system-prop`

Sets a system property of the JVM, for example `-Dmyprop=myvalue`. See Section 14.2, “Gradle properties and system properties”.

`-d`, `--debug`

Log in debug mode (includes normal stacktrace). See Chapter 18, *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 12, *Using the Gradle Graphical User Interface*.

`-I`, `--init-script`

Specifies an initialization script. See Chapter 61, *Initialization Scripts*.

`-i`, `--info`

Set log level to info. See Chapter 18, *Logging*.

`-m`, `--dry-run`

Runs the build with all task actions disabled. See Section 11.7, “Dry Run”.

`--offline`

Specifies that the build should operate without accessing network resources. See Section 51.9.2, “Command line options to override caching”.

`-P`, `--project-prop`

Sets a project property of the root project, for example `-Pmyprop=myvalue`. See Section 14.2, “Gradle properties and system properties”.

`-p`, `--project-dir`

Specifies the start directory for Gradle. Defaults to current directory. See Section 11.5, “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 57.9, “Decoupled Projects”).

`--parallel-threads (incubating)`

Build projects in parallel, using the specified number of executor threads. For example `--parallel-threads=`. This option should only be used with decoupled projects (see Section 57.9, “Decoupled Projects”).

`--profile`

Profiles build execution time and generates a report in the *`buildDir`*`/reports/profile` directory. See Section 11.6.7, “Profiling a build”.

`--project-cache-dir`

Specifies the project-specific cache directory. Default value is `.gradle` in the root project directory. See Section 14.6, “Caching”.

`-q`, `--quiet`
Log errors only. See Chapter 18, *Logging*.

`--recompile-scripts`
Specifies that cached build scripts are skipped and forced to be recompiled. See Section 14.6, “Caching”.

`--refresh-dependencies`
Refresh the state of dependencies. See Section 51.9.2, “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 18, *Logging*.

`-s`, `--stacktrace`
Print out the stacktrace also for user exceptions (e.g. compile error). See Chapter 18, *Logging*.

`-u`, `--no-search-upwards`
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 11.2, “Excluding tasks”.

The above information is printed to the console when you execute **`gradle -h`**.

D.1. Deprecated command-line options

`--no-color`
Do not use color in the console output. This option has been replaced by the `--console plain` option.

D.2. Daemon command-line options

The Chapter 19, *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 19, *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.3. 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 20.1, “Configuring the build environment via gradle.properties” contains specific information about Gradle configuration available via system properties.

D.4. 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=true"` to use the Gradle daemon without needing to use the `--daemon` option every time you run Gradle. Section 20.1, “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

Existing IDE Support and how to cope without it

E.1. IntelliJ

Gradle has been mainly developed with Idea IntelliJ and its very good Groovy plugin. Gradle's build script [28] has also been developed with the support of this IDE. IntelliJ allows you to define any filepattern to be interpreted as a Groovy script. In the case of Gradle you can define such a pattern for `build.gradle` and `settings.gradle`. This will already help very much. What is missing is the classpath to the Gradle binaries to offer content assistance for the Gradle classes. You might add the Gradle jar (which you can find in your distribution) to your project's classpath. It does not really belong there, but if you do this you have a fantastic IDE support for developing Gradle scripts. Of course if you use additional libraries for your build scripts they would further pollute your project classpath.

We hope that in the future `*.gradle` files get special treatment by IntelliJ and you will be able to define a specific classpath for them.

E.2. Eclipse

There is a Groovy plugin for eclipse. We don't know in what state it is and how it would support Gradle. In the next edition of this user guide we can hopefully write more about this.

E.3. Using Gradle without IDE support

What we can do for you is to spare you typing things like `throw new org.gradle.api.tasks.StopExecu` and just type `throw new StopExecutionException()` instead. We do this by automatically adding a set of import statements to the Gradle scripts before Gradle executes them. Listed below are the imports added to each script.

Figure E.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.component.*
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.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.plugins.sonar.*
import org.gradle.api.plugins.sonar.model.*
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.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.buildinit.plugins.*
import org.gradle.buildinit.tasks.*
import org.gradle.external.javadoc.*
import org.gradle.ide.cdt.*
import org.gradle.ide.cdt.tasks.*
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.platform.*
```

```
import org.gradle.jvm.plugins.*
import org.gradle.jvm.tasks.*
import org.gradle.jvm.toolchain.*
import org.gradle.language.*
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.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.objectivecpp.*
import org.gradle.language.objectivecpp.plugins.*
import org.gradle.language.objectivecpp.tasks.*
import org.gradle.language.rc.*
import org.gradle.language.rc.plugins.*
import org.gradle.language.rc.tasks.*
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.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.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.test.*
import org.gradle.play.*
import org.gradle.play.platform.*
import org.gradle.play.plugins.*
```

```
import org.gradle.play.tasks.*
import org.gradle.play.toolchain.*
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.sonar.runner.*
import org.gradle.sonar.runner.plugins.*
import org.gradle.sonar.runner.tasks.*
```

```
import org.gradle.testing.jacoco.plugins.*
import org.gradle.testing.jacoco.tasks.*
import org.gradle.util.*
```

---

[28] Gradle is built with Gradle

Gradle User Guide

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

??