Gradle User Guide

A better way to build

Version 1.0-milestone-8a

Hans Dockter

Adam Murdoch

Table of Contents

List of Examples

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 p and `ivy.xml` files.
- Ant tasks and builds as first class citizens.
- *Groovy* build scripts.
- A rich domain model for describing your build.

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

1.1. About this user guide

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

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). Thus avoiding 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 dirs 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 features compared to other declarative build systems. Our Groovy support is also not just some simple coating sugar layer. The whole Gradle API is fully groovynized. Only by that using Groovy is the fun and productivity gain it can be.

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 in case of *build scripts*. There are a couple of dynamic languages out there. Why Groovy? The answer lies in the context Gradle is operating in. Although Gradle is a general purpose build tool at its core, its main focus are Java projects. In such projects obviously the team members know Java. We think a build should be as transparent as possible to *all* team members.

You might argue why not using Java then as the language for build scripts. We think this is a valid question. It would have the highest transparency for your team and the lowest learning curve. But due to limitations of Java such a build language would not be as nice, expressive and powerful as it could be. [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 builds a lot on top of that. But on a common ground with Java.

For Java teams which share also Python or Ruby knowledge or are happy to learn it, the above arguments don't apply. 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 to be installed. Gradle requires a JDK 1.5 or higher. Gradle ships with its own Groovy library, therefore no Groovy needs to be installed. Any existing Groovy installation is ignored by Gradle.

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

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 the 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.

> **For Un*x users**
>
> You need a GNU compatible tool to unzip Gradle, if you want the file permissions to be

properly set. We mention this as some zip front ends for Mac OS X don't restore the file permissions properly.

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 gradle version and also local environment configuration (groovy and jvm version, 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 `GRADLE_OPTS` or `JAVA_OPTS`. Those variables can be used together. `JAVA_OPTS` is by convention an environment variable shared by many Java applications. A typical use case would be to set the HTTP proxy in `JAVA_OPTS` and the memory options in `GRADLE_OPTS`. Those variables can also be set at the beginning of the **gradle** or **gradlew** script.

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`). The Gradle Daemon is currently an experimental feature and may introduce build failures. More information about troubleshooting daemon is located in Chapter 13, *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 to 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*. A project represents some component of your software which can be built. What this means exactly depends on what it is that you are building. 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 `bui` 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. The first build script

`build.gradle`

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

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

Example 6.2. Execution of a build script

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

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

> **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. See Chapter 18, *Logging* for more details about the command-line options which affect Gradle's output.

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, well, you are right. Gradle tasks are the equivalent to Ant targets. But as you will see, they are much more powerful. We have used a different 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*.

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 expose to 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 dependencies between your tasks.

Example 6.6. Declaration of dependencies between tasks

build.gradle

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

Output of **gradle -q intro**

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

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

Example 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 16.4, "Adding dependencies to a task".

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

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*. This is different to Ant. For example you can create additional dependencies.

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 out of the box tasks provided by the plugins (e.g. `compile`).

6.9. Dynamic task properties

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

Example 6.12. Assigning properties to a task

`build.gradle`

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

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

Output of **`gradle -q showProps`**

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

6.10. Using Ant Tasks

Ant tasks are first-class citizens in Gradle. Gradle provides excellent integration for Ant tasks simply by relying on Groovy. Groovy is shipped with the fantastic `AntBuilder`. Using Ant tasks from Gradle is as convenient and more powerful than using Ant tasks from a `build.xml` file. From below example 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 19, *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 51, *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 describe in full detail later (See Chapter 47, *The Build Lifecycle*) Gradle has a configuration phase and an execution phase. After the configuration phase Gradle knows all tasks that should be executed. Gradle offers you a hook to make use of this information. A use-case for this would be to check if the release task is part of the tasks to be executed. Depending on this you can assign different values to some variables.

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

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

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 16, *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 C, *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` which is in both the binary and source distributions 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 Code-quality 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
    testCompile group: 'junit', name: 'junit', version: '4.+'
}
```

You can find out more in Chapter 8, *Dependency Management Basics*.

7.2.3. Customising the project

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

Example 7.5. Customization of MANIFEST.MF

build.gradle

```
sourceCompatibility = 1.5
version = '1.0'
jar {
    manifest {
        attributes 'Implementation-Title': 'Gradle Quickstart', 'Implementatio
    }
}
```

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

> **What properties are available?**
>
> You can use **`gradle propert`** to list the properties of a project. This will allow you to see the properties added by the Java plugin, and their default values.

Example 7.6. Adding a test system property

build.gradle

```
test {
    systemProperties 'property': 'value'
}
```

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 import your project into Eclipse, 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 on Eclipse task can be found in Chapter 35, *The Eclipse Plugin*.

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', 'Implementatio
    }
}

repositories {
    mavenCentral()
}

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

Note: The code for this example can be found at `samples/java/multiproject` which is in both the binary and source distributions of Gradle.

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

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 `se` . 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 48, *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:4.8.2'
    }

    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.

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 JAR produced by the `shared` project. Due to this dependency, Gradle will ensure that project s always gets built before project `api`.

Example 7.13. Multi-project build - dependencies between projects

`api/build.gradle`

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

See Section 48.7.1, “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 22, *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*.

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.Fi
    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 22.5, “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 42.3, "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.Fi
}
```

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.

There is a shortcut form for declaring external dependencies, which uses a string of the form "*grou*
.

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 42.4, "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 42.6, "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. In this instance, 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 43, *Publishing artifacts*.

8.7. Where to next?

For all the details of dependency resolution, see Chapter 42, *Dependency Management*, and for artifact publication see Chapter 43, *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 capabilties 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` which is in both the binary and source distributions 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 c 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 (configuration inherits this dependency, so the groovy libraries will be included in classpath when compiling Groovy and Java source. For our sample, we will use Groovy 1.6.0 from the public Maven repository:

Example 9.2. Dependency on Groovy 1.6.0

build.gradle

```
repositories {
    mavenCentral()
}

dependencies {
    groovy group: 'org.codehaus.groovy', name: 'groovy', version: '1.7.10'
}
```

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 {
    groovy group: 'org.codehaus.groovy', name: 'groovy', version: '1.7.10'
    testCompile group: 'junit', name: 'junit', version: '4.8.2'
}
```

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 23, *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 some of the Gradle's support for web applications. Gradle provides two plugins for web application developement: 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` which is in both the binary and source distributions 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/weba` . Your compiled classes, and their runtime dependencies are also included in the WAR file.

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 will be added to the WAR file for you.

Example 10.2. Running web application with Jetty plugin

build.gradle

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

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 25, *The War Plugin* and the Jetty plugin in Chapter 27, *The Jetty Plugin*. You can find more sample Java projects in the `samples/webApplic` directory in the Gradle distribution.

11

Using the Gradle Command-Line

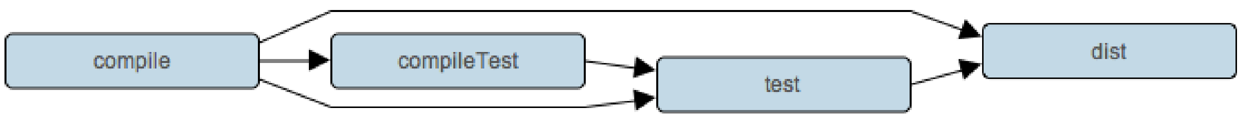
This chapter introduces the basics of the Gradle command-line. You run a build using the **gradle** command, which you have already seen in action in previous chapters.

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 it a dependency of another task, or both. Let's look at an example.

Below four tasks are defined. Both `dist` and `test` depend on the `compile` task. Running **`gradle`** 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
```

Because each task is executed once only, executing **`gradle test test`** is exactly the same as executing **`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 `cc` are not executed either. Those dependencies of `test` that are required by another task, such as `cc` , are still executed.

11.3. Task name abbreviation

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

Example 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.

11.4. Selecting which build to execute

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

Example 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:

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'.
```

11.5. 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.

11.5.1. 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'
```

11.5.2. 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

Help tasks
----------
dependencies - Displays the dependencies of root project 'projectReports'.
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 o

To see all tasks and more detail, run with --all.
```

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

Help tasks
----------
dependencies - Displays the dependencies of root project 'projectReports'.
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 o
```

11.5.3. 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 11.12. 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:1.8.4 [default]

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

compile
+--- projectReports:api:1.0-SNAPSHOT [compile]
|    \--- org.codehaus.groovy:groovy-all:1.8.4 [default]
\--- commons-io:commons-io:1.2 [default]
```

11.5.4. 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 11.13. Information about properties

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

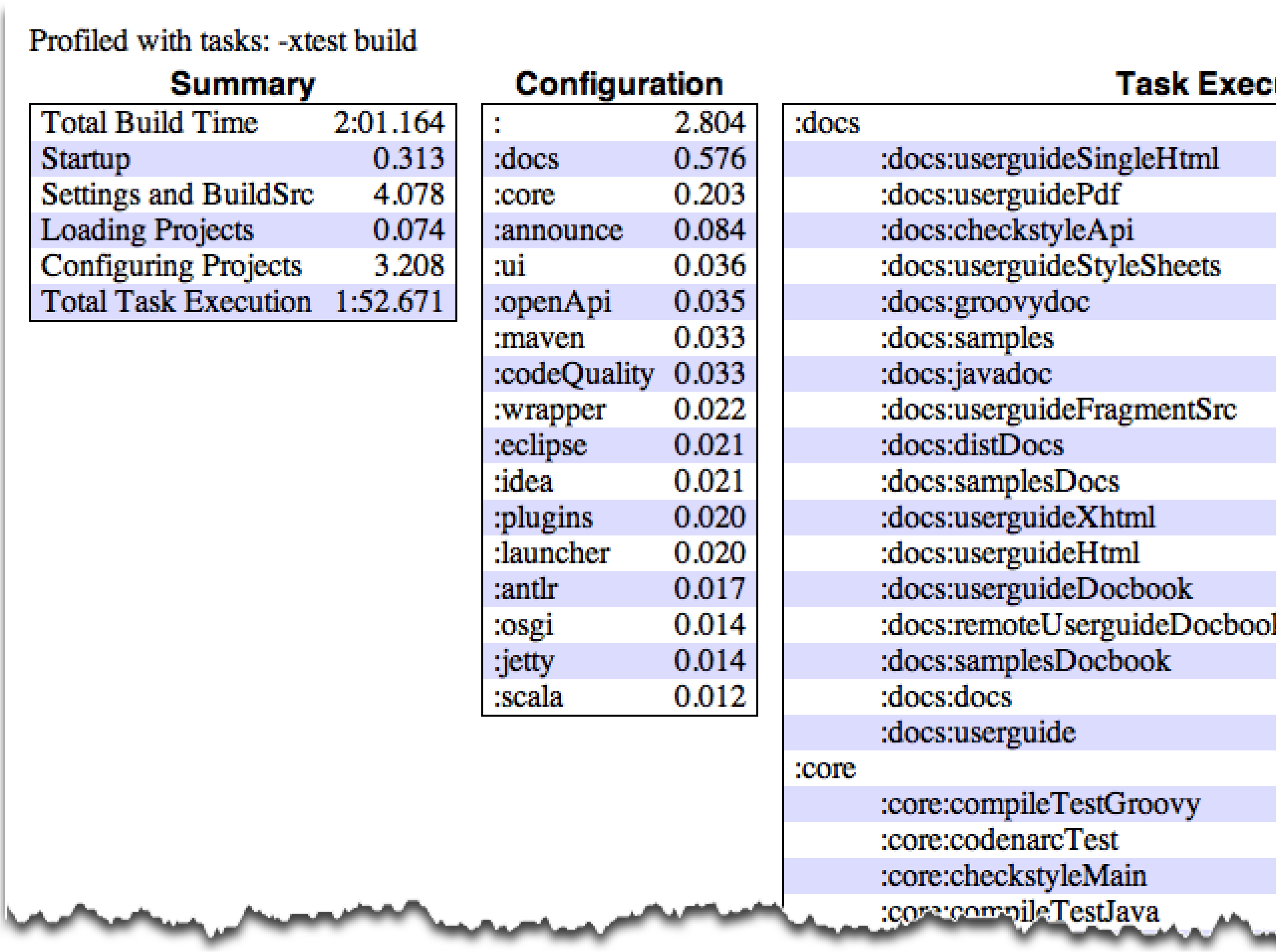
```
> gradle -q api:properties
------------------------------------------------------------
Project :api - The shared API for the application
------------------------------------------------------------

additionalProperties: {}
allprojects: [project ':api']
ant: org.gradle.api.internal.project.DefaultAntBuilder@12345
antBuilderFactory: org.gradle.api.internal.project.DefaultAntBuilderFactory@123
artifacts: org.gradle.api.internal.artifacts.dsl.DefaultArtifactHandler@12345
asDynamicObject: org.gradle.api.internal.DynamicObjectHelper@12345
buildDir: /home/user/gradle/samples/userguide/tutorial/projectReports/api/build
buildDirName: build
buildFile: /home/user/gradle/samples/userguide/tutorial/projectReports/api/buil
```

11.5.5. 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 `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 Exec

:docs	
	:docs:userguideSingleHtml
	:docs:userguidePdf
	:docs:checkstyleApi
	:docs:userguideStyleSheets
	:docs:groovydoc
	:docs:samples
	:docs:javadoc
	:docs:userguideFragmentSrc
	:docs:distDocs
	:docs:samplesDocs
	:docs:userguideXhtml
	:docs:userguideHtml
	:docs:userguideDocbook
	:docs:remoteUserguideDocbool
	:docs:samplesDocbook
	:docs:docs
	:docs:userguide
:core	
	:core:compileTestGroovy
	:core:codenarcTest
	:core:checkstyleMain
	:core:compileTestJava

11.6. Dry Run

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

11.7. Summary

In this chapter, you have seen some of the things you can do with Gradle from the command-line. You can find out more about the **gradle** command in Appendix C, *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.

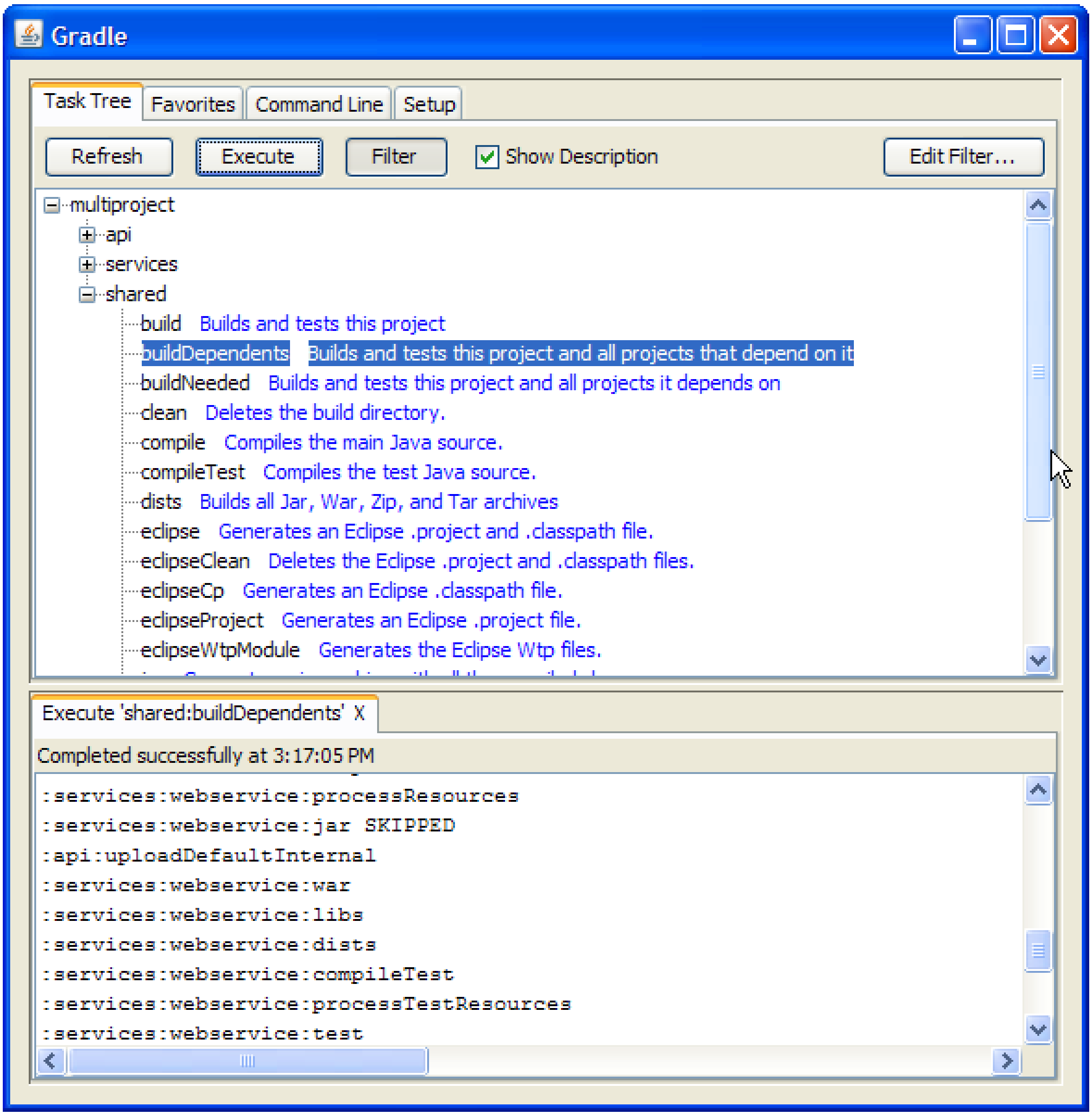
Example 12.1. Launching the GUI

```
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 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.

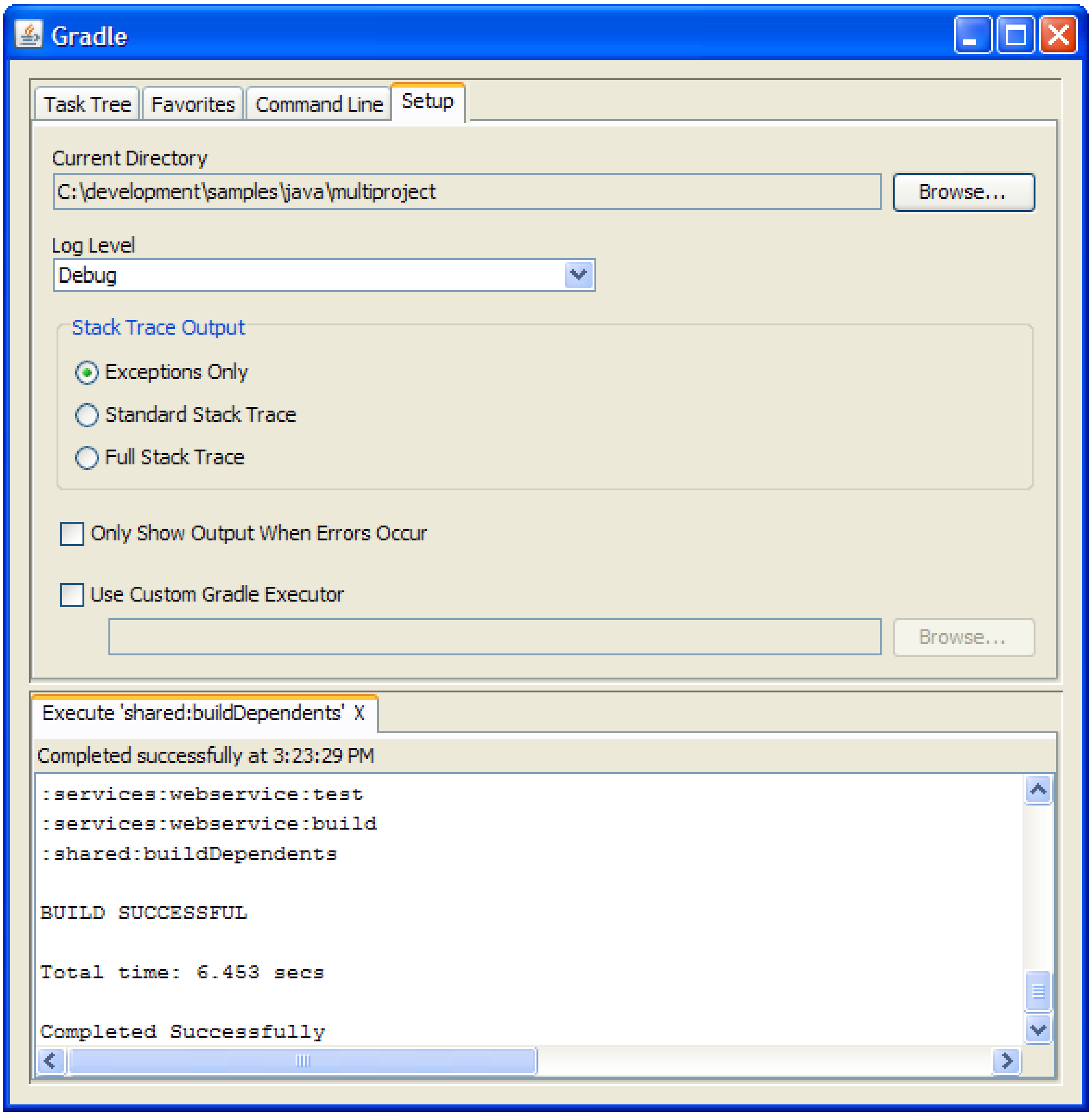
12.3. Command Line

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

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 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

The Gradle Daemon

13.1. 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 -t is executed a number of times.

For above sorts of 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 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 it's going to be hard for you to go back to regular use of Gradle. At the moment, we marked the Gradle build daemon as experimental because there are still many aspects of the daemon we'd like to improve.

The Tooling API (see Chapter 54, *Embedding Gradle*) uses the daemon all the time, e.g. you cannot officially use the Tooling API without the daemon. This means that if you use the STS Gradle plugin for Eclipse or new Intellij IDEA plugin (IDEA>10) the daemon acts behind the hood.

In future the daemon will offer more features:

- Snappy up-to-date checks: use native file system change notifications (eg 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.
- Preemtive execution of certain tasks, for example compilation. Quicker feedback.
- Fast and accurate bash tab completion.
- Periodically garbage collect the Gradle caches.

13.2. Implementation notes

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.

Here're 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 build with a different java home then a brand new deamon will be forked.
- At the moment daemon is coupled with 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 deamon will be started. This also have a consequence for the `--stop` command line instruction: You can only stop daemons that were started with the gradle version you use when running `--stop`.

We plan to improve the ways of managing / pooling the daemons in future.

13.3. Usage and troubleshooting

For command line usage take a look dedicated section in Appendix C, *Gradle Command Line*. If you are tired of using the same command line options again and again, take a look at Section 14.3, “Configuring gradle via gradle.properties”. The section contains information on how to configure certain behavior of the daemon (including turning on the daemon by default) in a more 'persistent' way.

As mentioned earlier we are actively improving the daemon. At the moment the daemon is marked as 'experimental' in the user interface. We encourage everyone to try the daemon out and get back to us with feedback (or even better: the pull requests). 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 is located by default in the `gradle user home`.
- You may want to start the daemon in `--foreground` mode to observe how the build is executed.

13.4. Daemon properties

Some daemon settings can be configured in `gradle.properties`. For example, jvm args - memory settings or the java home. Please find more information in Section 14.2, "Gradle properties and system properties"

14

Tutorial - 'This and That'

14.1. Directory creation

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

Example 14.1. Directory creation with mkdir

`build.gradle`

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

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

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

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

Example 14.2. Directory creation with Directory tasks

build.gradle

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

Output of **`gradle -q otherResources`**

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

A *Directory Task* is a simple task whose name is a relative path to the project dir [5] . During the execution phase the directory corresponding to this path gets created if it does not exist yet. Another interesting thing to note in this example, is that you can also pass tasks objects to the dependsOn declaration of a task.

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

You can also add properties directly to your project object via the `-P` command line option. For more exotic use cases you can even pass properties *directly* to the project object via system and environment properties. For example if you run a build on a continuous integration server where you have no admin rights for the *machine*. Your build script needs properties which values should not be seen by others. Therefore you can't use the `-P` option. In this case you can add an environment property in the project administration section (invisible to normal users). [6] If the environment property follows the pattern `ORG_GRADLE_PROJECT_*propertyName*=somevalue`, p is added to your project object. If in the future CI servers support Gradle directly, they might start Gradle via its main method. Therefore we already support the same mechanism for system properties. The only difference is the pattern, which is `org.gradle.project.*propertyName*`.

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

Example 14.3. Setting properties with a gradle.properties file

`gradle.properties`

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

`build.gradle`

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

Output of **`gradle -q -PcommandLineProjectProp=commandLineProjectPropValue -Do`**

```
> gradle -q -PcommandLineProjectProp=commandLineProjectPropValue -Dorg.gradle.p
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. In case this property does not exists, an exception is thrown and the build fails. If your build script relies on optional properties the user might set for example in a gradle.properties file, you need to check for existence before you can access them. You can do this by using the method `hasProperty('propertyName')` which returns `true` or `false`.

14.3. Configuring gradle via gradle.properties

The idea is that certain settings like jvm memory settings, java home, deamon on/off are best useful if they can versioned with the project in your VCS so that the entire team can work with consistent environment. You can place those settings in the `gradle.properties`. The configuration is applied in following order (in case an option is configured in multiple locations the last one wins):

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

The following properties can be used to configure gradle:

`org.gradle.daemon`
: When set to `true` the gradle daemon is to run the build. This is our favorite property because we nearly always run gradle jobs with the daemon.

`org.gradle.jvmargs`
: Specifies the jvmargs used for the daemon process. In future this setting will be honored by regular gradle builds. Currently, only daemon uses it. The setting is particularly useful for tweaking memory settings. At the moment the default settings are pretty generous with regards to memory.

`org.gradle.java.home`
: Specifies the java home for the daemon process. In future this setting will be honored by regular gradle builds. Currently, only daemon uses it. The value can be set to either `jdk` or `j` location, however, depending on what does your build do, `jdk` is safer. Reasonable default is used if the setting is unspecified.

14.4. Accessing the web via a proxy

Setting a proxy for web access (for example for downloading dependencies) is easy. Gradle does not need to provide special functionality for this. The JVM can be instructed to go via proxy by setting certain system properties. You could set these system properties directly in your build script with `System.properties['proxy.proxyUser'] = 'userid'`. An arguably nicer way is shown in Section 14.2, “Gradle properties and system properties”. Your gradle.properties file could look like this:

Example 14.4. Accessing the web via a proxy

`gradle.properties`

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

We could not find a good overview for all possible proxy settings. One place to look are the constants in a file from the ant project. Here a link to the svn view. The other is a Networking Properties page from the JDK docs. If anyone knows a better overview please let us know via the mailing list.

14.4.1. 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.

14.5. 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 14.5. 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
```

14.6. Configuring arbitrary objects

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

Example 14.6. Configuring arbitrary objects

`build.gradle`

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

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

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

14.7. Configuring arbitrary objects using an external script

You can also configure arbitrary objects using an external script.

Example 14.7. Configuring arbitrary objects using a script

`build.gradle`

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

`other.gradle`

```
beginIndex = 1;
endIndex = 5;
```

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

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

14.8. 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 `-C rebuild` 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.

---

[5] The notation `dir('/somepath')` is a convenience method for `tasks.add('somepath', ty`

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

15

Writing Build Scripts

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

15.1. 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.

15.2. 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 creates an instance of type `Project` and associates this `Project` object with the build script. As the build script executes, it configures this `Project` object:

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

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

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

Example 15.1. Accessing property of the Project object

`build.gradle`

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

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

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

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

15.2.1. 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 15.1. 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

15.3. 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.

15.4. 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 customs plugins and tasks.

15.4.1. Groovy JDK

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

Example 15.2. 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.

15.4.2. Property accessors

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

Example 15.3. Property accessors

build.gradle

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

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

15.4.3. Optional parentheses on method calls

Parentheses are optional for method calls.

Example 15.4. Method call without parentheses

build.gradle

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

15.4.4. List and map literals

Groovy provides some shortcuts for defining `List` and `Map` instances.

Example 15.5. 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
apply plugin: 'java'

Map<String, String> map = new HashMap<String, String>()
map.put('plugin', 'java')
apply(map)
```

15.4.5. 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 15.6. Closure as method parameter

`build.gradle`

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

15.4.6. 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 15.7. Closure delegates

`build.gradle`

```
dependencies {
    assert delegate == project.dependencies
    compile('junit:junit:4.8.2')
    delegate.compile('junit:junit:4.8.2')
}
```

16

More about Tasks

In the introductory tutorial (Chapter 6, *Build Script Basics*) you have learned how to create simple tasks. You have also learned how to add additional behavior to these tasks later on. And you have learned how to create dependencies between tasks. This was all about simple tasks. But Gradle takes the concept of tasks further. Gradle supports *enhanced tasks*, that is, tasks which have their own properties and methods. This is really different to what you are used to with Ant targets. Such enhanced tasks are either provided by you or are provided by Gradle.

16.1. 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 16.1. 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 16.2. Defining tasks - using strings

build.gradle

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

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

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

Example 16.3. Defining tasks with alternative syntax

build.gradle

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

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

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

16.2. Locating tasks

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

Example 16.4. Accessing tasks as properties

build.gradle

```
task hello

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

Tasks are also available through the `tasks` collection.

Example 16.5. 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 16.6. 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.

16.3. 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 16.7. Creating a copy task

`build.gradle`

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

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

Example 16.8. Configuring a task - various ways

build.gradle

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

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

There is a more convenient way of doing this.

Example 16.9. Configuring a task - fluent interface

build.gradle

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

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

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

Example 16.10. Configuring a task - with closure

build.gradle

```
task myCopy(type: Copy)

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

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

There is a slightly different ways of doing this.

Example 16.11. Configuring a task - with configure() method

`build.gradle`

```
task myCopy(type: Copy)

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

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

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

Example 16.12. Defining a task with closure

`build.gradle`

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

16.4. 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. Below is an example which adds a dependency from `projectA:taskX` to `projectB:taskY`:

Example 16.13. Adding dependency on task from another project

build.gradle

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

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

Output of **gradle -q taskX**

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

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

Example 16.14. Adding dependency using task object

build.gradle

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

task taskY << {
    println 'taskY'
}

taskX.dependsOn taskY
```

Output of **gradle -q taskX**

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

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

Example 16.15. Adding dependency using closure

`build.gradle`

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

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

task lib1 << {
    println 'lib1'
}

task lib2 << {
    println 'lib2'
}

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

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

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

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

16.5. Adding a description to a task

You can add a description to your task. This description is for example displayed when executing **g**
.

Example 16.16. Adding a description to a task

`build.gradle`

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

16.6. Replacing tasks

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

Example 16.17. Overwriting a task

`build.gradle`

```
task copy(type: Copy)

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

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

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

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

16.7. Skipping tasks

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

16.7.1. Using a predicate

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

Example 16.18. Skipping a task using a predicate

`build.gradle`

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

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

Output of **`gradle hello -PskipHello`**

```
> gradle hello -PskipHello
:hello SKIPPED

BUILD SUCCESSFUL

Total time: 1 secs
```

16.7.2. Using StopExecutionException

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

Example 16.19. Skipping tasks with StopExecutionException

`build.gradle`

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

compile.doFirst {
    // Here you would put arbitrary conditions in real life. But we use this a
    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]

16.7.3. Enabling and disabling tasks

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

Example 16.20. Enabling and disabling tasks

`build.gradle`

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

Output of **`gradle disableMe`**

```
> gradle disableMe
:disableMe SKIPPED

BUILD SUCCESSFUL

Total time: 1 secs
```

16.8. 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.

16.8.1. 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 16.21. A generator task

`build.gradle`

```
task transform {
    srcFile = file('mountains.xml')
    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 16.22. Declaring the inputs and outputs of a task

`build.gradle`

```
task transform {
    srcFile = file('mountains.xml')
    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`.

16.8.2. 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 takes note of any files created, changed or deleted in the output directories of the task. Gradle persists both snapshots for 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 next time the task is executed.

16.9. Task rules

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

Example 16.23. Task rule

build.gradle

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

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

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

The String parameter is used as a description for the rule. This description is shown when running for example **`gradle tasks`**.

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

Example 16.24. 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
```

16.10. Summary

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

---

[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 D, *Existing IDE Support and how to cope without it*).

17

Working With Files

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

17.1. Locating files

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

Example 17.1. 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. The supplied object's `toString()` value is used as the file path. If this path is an absolute path, it is used to construct a `File` instance. Otherwise, a `File` instance is constructed by prepending the project directory path to the supplied path. The `file()` method also understands URLs, such as `file:/s`
.

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.

17.2. 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 for the `file()` method, described in Section 17.1, “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 17.2. Creating a file collection

`build.gradle`

```
FileCollection collection = files('src/file1.txt', new File('src/file2.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 17.3. 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 17.4. 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.

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.

17.3. 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 `FileTr` 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 sourc
.

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 17.5. 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 17.6. 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"
}
```

17.4. 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 `FileTre` 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 17.7. 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'))
```

17.5. Specifying a set of input files

Many objects in Gradle have properties which accept a set of input files. For example, the `Compile` task has a `source` property, which defines the source files to compile. You can set the value of this property using any of the types supported by the files() method, which we have seen in 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 17.8. 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 { z
    }
}
```

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 17.9. 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() }
}
```

17.6. 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 to map the files 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, you use the `CopySpec.into()` method.

Example 17.10. 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. The `into()` accepts any of the arguments that the file() method does. Here is another example:

Example 17.11. 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 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 17.12. 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.t
}
```

You can also use the `Project.copy()` method to copy files. It works the same way as the task.

Example 17.13. Copying files using the copy() method

build.gradle

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

17.6.1. Renaming files

Example 17.14. Renaming files as they are copied

build.gradle

```
task rename(type: Copy) {
    from 'src/main/webapp'
    into 'build/explodedWar'
    // Use a closure to map the file name
    rename { String fileName ->
        fileName.replace('-staging-', '')
    }
    // Use a regular expression to map the file name
    rename '(.+)-staging-(.+)', '$1$2'
    rename(/(.+)-staging-(.+)/, '$1$2')
}
```

17.6.2. Filtering files

Example 17.15. Filtering files as they are copied

build.gradle

```
import org.apache.tools.ant.filters.FixCrLfFilter
import org.apache.tools.ant.filters.ReplaceTokens

task filter(type: Copy) {
    from 'src/main/webapp'
    into 'build/explodedWar'
    // Substitute property references 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]"
    }
}
```

17.6.3. Using the CopySpec class

Copy specs form a hierarchy. A copy spec inherits its destination path, include patterns, exclude patterns, copy actions, name mappings, filters.

Example 17.16. Nested copy specs

`build.gradle`

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

17.7. 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/li` directory.

Example 17.17. Using the Sync task to copy dependencies

`build.gradle`

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

17.8. Creating archives

A project can have as many as 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`, and `War`. They all work the same way, so let's look at how you create a ZIP file.

Example 17.18. Creating a ZIP archive

`build.gradle`

```
apply plugin: 'java'

task zip(type: Zip) {
    from 'src/dist'
    into('libs') {
        from configurations.runtime
    }
}
```

The archive tasks all work exactly the same way as the `Copy` task, and implement the same `CopySpec` interface. As with the `Copy` task, you specify the input files using the `from()` method, and can optionally specify where they end up in the archive using the `into()` method. You can filter the contents of file, rename files, and all the other things you can do with a copy spec.

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.

17.8.1. Archive naming

The default name for a generated archive is *`projectName-version.type`* For example:

Example 17.19. Creation of ZIP archive

`build.gradle`

```
apply plugin: 'java'

version = 1.0

task myZip(type: Zip) {
    from 'somedir'
}

println myZip.archiveName
println relativePath(myZip.destinationDir)
println relativePath(myZip.archivePath)
```

Output of **`gradle -q myZip`**

```
> gradle -q myZip
zipProject-1.0.zip
build/distributions
build/distributions/zipProject-1.0.zip
```

This adds a `Zip` archive task with the name `myZip` which produces ZIP file`zipProject-1.0.zip` . It is important to distinguish between the name of the archive task and the name of the archive generated by the archive task. The default name for archives can be changed with the `archivesBa` 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 17.1, “Archive tasks - naming properties”. You can, for example, change the name of the archive:

Example 17.20. Configuration of archive task - custom archive name

`build.gradle`

```
apply plugin: 'java'
version = 1.0

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

println myZip.archiveName
```

Output of **`gradle -q myZip`**

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

You can further customize the archive names:

Example 17.21. Configuration of archive task - appendix & classifier

`build.gradle`

```
apply plugin: 'java'
archivesBaseName = 'gradle'
version = 1.0

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

println myZip.archiveName
```

Output of **`gradle -q myZip`**

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

Table 17.1. 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 file 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 are generated into *`project.buildDir/libraries`*. ZIPs and TARs are generated into *`project.buildDir/dis`*.	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.

17.8.2. Sharing content between multiple archives

Using 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 43, *Publishing artifacts*

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 the 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`	QUIET and higher
`-i`	INFO and higher
`-d`	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 loglevel option `-d` is chosen, truncated stacktraces are always printed.
`-s`	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`	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. 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'
    }
}
```

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 47.6, “Responding to the lifecycle in the build script”.

- `BuildListener`
- `ProjectEvaluationListener`

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

19

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 , 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.

19.1. Using Ant tasks and types in your build

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

You execute an Ant task by calling a method on the `AntBuilder` instance. You use the task name as the method name. For example, you execute the Ant `echo` task by calling the `ant.echo(`

method. The attributes of the Ant task are passed as Map parameters to the method. Below is an example which executes the `echo` task. Notice that we can also mix Groovy code and the Ant task markup. This can be extremely powerful.

Example 19.1. 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 19.2. 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 19.3. 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 19.4. 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

19.1.1. Using custom Ant tasks in your build

To make custom tasks available in your build, you can use the `taskdef` (usually easier) or `typede` 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 19.5. Using a custom Ant task

build.gradle

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

You can use Gradle's dependency management to assemble the classpath to use for the custom tasks. To do this, you need to define a custom configuration for the classpath, then add some dependencies to the configuration. This is described in more detail in Section 42.4, “How to declare your dependencies”.

Example 19.6. 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 19.7. Using a custom Ant task and dependency management together

build.gradle

```
task check << {
    ant.taskdef(name: 'pmd', classname: 'net.sourceforge.pmd.ant.PMDTask', clas
    ant.pmd(shortFilenames: 'true', failonruleviolation: 'true', rulesetfiles:
        formatter(type: 'text', toConsole: 'true')
        fileset(dir: 'src')
    }
}
```

19.2. 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 19.8. 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 19.9. 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 19.10. 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 19.11. 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
```

19.3. 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 19.12. Setting an Ant property

`build.gradle`

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

`build.xml`

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

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

Example 19.13. Getting an Ant property

`build.xml`

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

`build.gradle`

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

There are several ways to set an Ant reference:

Example 19.14. Setting an Ant reference

`build.gradle`

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

`build.xml`

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

There are several ways to get an Ant reference:

Example 19.15. Getting an Ant reference

`build.xml`

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

`build.gradle`

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

19.4. 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

20

Using Plugins

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

20.1. Declaring plugins

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

Example 20.1. Using a plugin

`build.gradle`

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

in your script. That's all. From a technological point of view plugins use just the same operations as you can use from your build scripts. That is, they use the Project and Task API. The Gradle plugins generally use this API to:

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

Let's check this out:

Example 20.2. Applying a plugin by id

`build.gradle`

```
apply plugin: 'java'

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

Output of **`gradle -q show`**

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

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

The `apply()` method either takes a string or a class as an argument. You can write

Example 20.3. Applying a plugin by type

`build.gradle`

```
apply plugin: org.gradle.api.plugins.JavaPlugin
```

Thanks to Gradle's default imports (see Appendix D, *Existing IDE Support and how to cope without it*) you can also write in this case.

Example 20.4. Applying a plugin by type

`build.gradle`

```
apply plugin: JavaPlugin
```

Any class, which implements the `Plugin` interface, can be used as a plugin. Just pass the class as an argument. You don't need to configure anything else for this.

If you want to use your own plugins, you must make sure that they are accessible via the build script classpath (see Chapter 51, *Organizing Build Logic* for more information). To learn more about how to write custom plugins, see Chapter 50, *Writing Custom Plugins*.

20.2. Using the convention object

If you use the Java plugin for example, there are a `compileJava` and a `processResources` task for your production code (the same is true for your test code). What if you want to change the default configuration? Let's try:

Example 20.5. Configuring a plugin task

`build.gradle`

```
apply plugin: 'java'

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

Output of **`gradle -q show`**

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

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

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

Example 20.6. Plugin convention object

`build.gradle`

```
apply plugin: 'java'

task show << {
    sourceSets.main.output.classesDir = new File(buildDir, 'output/classes')
    sourceSets.main.output.resourcesDir = new File(buildDir, 'output/resources

    println relativePath(compileJava.destinationDir)
    println relativePath(processResources.destinationDir)
}
```

Output of **`gradle -q show`**

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

The Java plugin has added a convention object with a `sourceSets` property, which we use to set the classes directory.

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

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

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

Example 20.7. Using the plugin convention object

`build.gradle`

```
apply plugin: 'java'

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

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

Output of **`gradle -q show`**

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

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

20.2.1. Declaring plugins multiple times

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

Example 20.8. Explicit application of an implied plugin

`build.gradle`

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

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

21

Standard Gradle plugins

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

21.1. Language plugins

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

Table 21.1. 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.

21.2. Experimental language plugins

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

Table 21.2. Language plugins

Plugin Id	Automatically applies	Works with	Description
`cpp`	-	-	Adds C++ source compilation capabilities to a project.
`cpp-exe`	`cpp`	-	Adds C++ executable compilation and linking capabilities to a project.
`cpp-lib`	`cpp`	-	Adds C++ library compilation and linking capabilities to a project.

21.3. Integration plugins

These plugins provide some integration with various build and runtime technologies.

Table 21.3. Integration plugins

Plugin Id	Automatically applies	Works with	Description
`announce`	-	-	Publish messages to your favourite platforms, such as Twitter or Growl.
`application`	`java`	-	Adds tasks for running and bundling a project as application.
`build-announcements`	announce	-	Sends local announcements to your desktop about interesting events in the build lifecycle.
`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 deploying 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*.

21.4. Software development plugins

These plugins provide help with your software development process.

Table 21.4. Software development plugins

Plugin Id	Automatically applies	Works with	Description

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`	-	-	Provides integration with the Sonar code quality platform.

21.5. 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 21.5. 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.

21.6. Third party plugins

You can find a list of external plugins on the wiki.

22

The Java Plugin

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

22.1. Usage

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

Example 22.1. Using the Java plugin

`build.gradle`

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

22.2. 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 unit test source code, which is compiled and executed using JUnit or TestNG.

22.3. Tasks

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

Table 22.1. 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.	`Compile`	Compiles production Java source files using javac.
`processResources`	-	`Copy`	Copies production resources into the production classes directory.
`classes`	`compileJava` and `processResources`. Some plugins add additional compilation tasks.	`Task`	Assembles the production classes directory.
`compileTestJava`	`compile`, plus all tasks which produce the test compile classpath.	`Compile`	Compiles test Java source files using javac.
`processTestResources`	-	`Copy`	Copies test resources into the test classes directory.
`testClasses`	`compileTestJava` and `processTestResources`. 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 the artifacts in the `archives` configuration, including the JAR file.

`clean`	-	`Delete`	Deletes the project build directory.
`clean`*`TaskName`*	-	`Delete`	Deletes the output files produced by the specified task. For example `cleanJa` 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 22.2. Java plugin - source set tasks

Task name	Depends on	Type	Description
`compile`*`SourceSet`*`Java`	All tasks which produce the source set's compile classpath.	`Compile`	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`	`compile`*`SourceSet`*`Java` and `process`*`SourceSet`*`Resources`. Some plugins add additional compilation tasks for the source set.	`Task`	Assembles the given source set's classes directory.

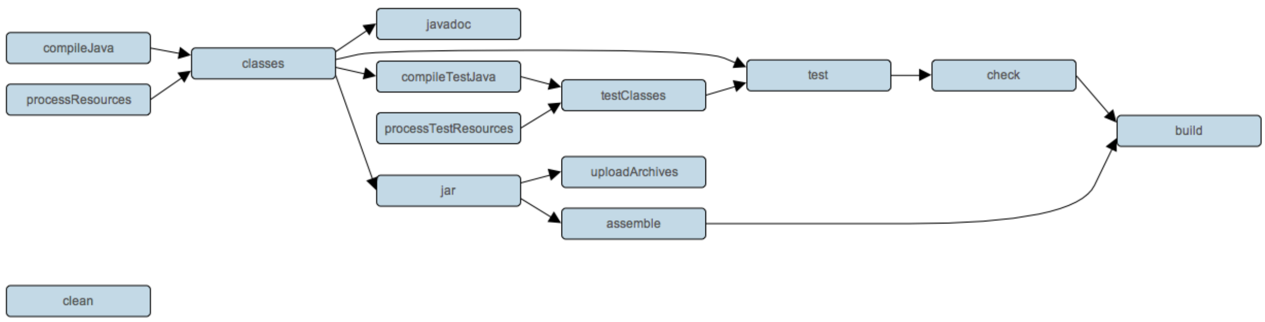
The Java plugin also adds a number of tasks which form a lifecycle for the project:

Table 22.3. 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 `build` 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 `build` 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`*`ConfigurationName`*	The tasks which produce the artifacts in configuration *`ConfigurationName`*.	`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`*`ConfigurationName`*	The tasks which uploads the artifacts in configuration *`ConfigurationName`*.	`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 22.1. Java plugin - tasks



22.4. 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 22.4. 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

22.4.1. 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 22.2. Custom Java source layout

`build.gradle`

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

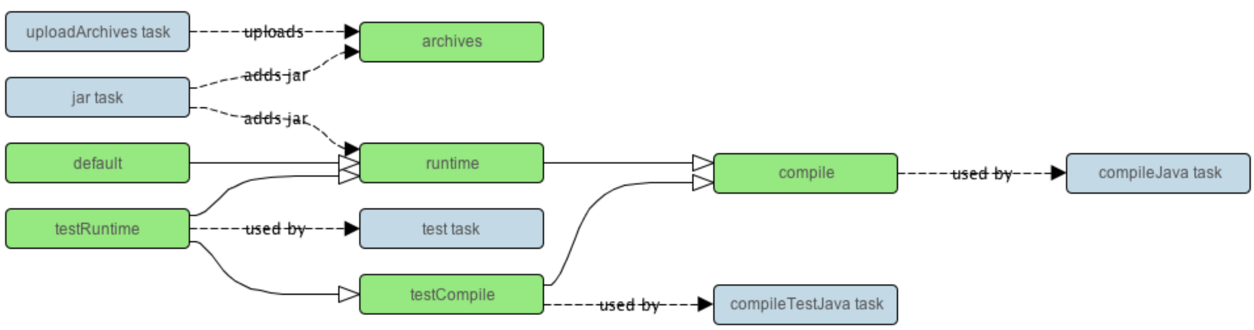
22.5. Dependency management

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 22.5. 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 22.2. Java plugin - dependency configurations



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

Table 22.6. 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 time dependencies for the given source set

22.6. 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 Section 20.2, “Using the convention object”).

Table 22.7. 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 22.8. Java plugin - other properties

Property name	Type	Default value	Descriptio
`sourceSets`	`SourceSetContainer` (read-only)	Not null	Contains tl project's source set
`sourceCompatibility`	`JavaVersion`. Can also set using a String or a Number, eg `'1.5'` or `1.5`.	`1.5`	Java versi compatibili to use whe compiling Java sourc
`targetCompatibility`	`JavaVersion`. Can also set using a String or Number, eg `'1.5'` or `1.5`.	*`sourceCompatibility`*	Java versi to generat classes fo
`archivesBaseName`	`String`	*`projectName`*	The basename use for archives, such as JA or ZIP files
`manifest`	`Manifest`	an empty manifest	The manife to include all JAR file
`metaInf`	`List`	`[]`	A set of file which spec the files to include in t directory o all JAR file

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

22.7. 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 22.3. 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 22.4. Configuring the source directories of a source set

build.gradle

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

22.7.1. 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 22.9. 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 17.5, “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 17.5, “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.

22.7.2. Defining new source sets

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

Example 22.5. 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 22.6, “Java plugin - source set dependency configurations”. You can use these configurations to define the compile and runtime dependencies of the source set.

Example 22.6. Defining source set dependencies

`build.gradle`

```
sourceSets {
    intTest
}

dependencies {
    intTestCompile 'junit:junit:4.8.2'
    intTestRuntime 'asm:asm-all:3.3.1'
}
```

The Java plugin also adds a number of tasks which assemble the classes for the source set, as shown in Table 22.2, “Java plugin - source set tasks”. For example, for a source set called `intTest`, you can run **`gradle intTestClasses`** to compile the int test classes.

Example 22.7. Compiling a source set

Output of **`gradle intTestClasses`**

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

BUILD SUCCESSFUL

Total time: 1 secs
```

22.7.3. Some source set examples

Adding a JAR containing the classes of a source set:

Example 22.8. Assembling a JAR for a source set

`build.gradle`

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

Generating Javadoc for a source set:

Example 22.9. 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 22.10. Running tests in a source set

build.gradle

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

22.8. 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 22.10. Java plugin - Javadoc properties

Task Property	Type	Default Value
`classpath`	`FileCollection`	`sourceSets.main.classes + sourceSets.ma`
`source`	`FileTree`. Can set using anything described in Section 17.5, “Specifying a set of input files”.	`sourceSets.main.allJava`
`destinationDir`	`File`	*`docsDir`*`/javadoc`
`title`	`String`	The name and version of the project

22.9. Clean

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

Table 22.11. Java plugin - Clean properties

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

22.10. 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 17.6, "Copying files".

Table 22.12. Java plugin - ProcessResources properties

Task Property	Type	Default Value
`srcDirs`	`Object`. Can set using anything described in Section 17.5, "Specifying a set of input files".	*`sourceSet`*`.resources`
`destinationDir`	`File`. Can set using anything described in Section 17.1, "Locating files".	*`sourceSet`*`.output.resou`

22.11. CompileJava

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

Table 22.13. Java plugin - Compile properties

Task Property	Type	Default Value
`classpath`	`FileCollection`	*`sourceSet`*`.compileCla`
`source`	`FileTree`. Can set using anything described in Section 17.5, "Specifying a set of input files".	*`sourceSet`*`.java`
`destinationDir`	`File`.	*`sourceSet`*`.output.cla`

22.12. Test

The `test` task is an instance of `Test`. It automatically detects and executes all unit tests in the `tes` 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.

22.12.1. Test execution

Tests are executed in a separate isolated JVM. 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. The task also provides a `debug` property, which when set to true, starts the test process in debug mode, suspended and listening on port 5005. This makes it very easy to debug your tests. You may also enable this using a system property as specified below.

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 `for` 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. Test 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.

22.12.2. System properties

There are two system properties that can affect test execution. Both of these are based off of the name of the test task with a suffix.

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 then 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`

Setting a system property of `taskName.debug` will run the tests in debug mode, suspended and listening on port 5005. For example: `gradle test -Dtest.single=ThisUniquelyNamedTes`

22.12.3. 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 scan up the inheritance tree into jar files on the test classpath.

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

22.12.4. Convention values

Table 22.14. 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`

22.13. 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 17.8, "Creating archives" and artifact configurations in Chapter 43, *Publishing artifacts*.

22.13.1. 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 22.11. Customization of MANIFEST.MF

`build.gradle`

```
jar {
    manifest {
        attributes("Implementation-Title": "Gradle", "Implementation-Version":
    }
}
```

You can create stand alone instances of a `Manifest`. You can use that for example, to share manifest information between jars.

Example 22.12. Creating a manifest object.

`build.gradle`

```
sharedManifest = manifest {
    attributes("Implementation-Title": "Gradle", "Implementation-Version": vers
}
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 22.13. 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.tx
            eachEntry { details ->
                if (details.baseValue != details.mergeValue) {
                    details.value = baseValue
                }
                if (details.key == 'foo') {
                    details.exclude()
                }
            }
        }
    }
}
```

Manifest are merged in the order they are declared by the `from` statement. If the based 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 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 22.14. Separate MANIFEST.MF for a particular archive

`build.gradle`

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

22.13.2. MetaInf

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

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

22.14. Uploading

How to upload your archives is described in Chapter 43, *Publishing artifacts*.

23

The Groovy Plugin

The Groovy plugin extends the Java plugin to add support for Groovy projects. It can deal with Groovy-only projects and with mixed Java/Groovy projects. It can even deal with Java-only projects. [9] The Groovy plugin supports joint compilation of Java and Groovy. This means that your project can contain Groovy classes which use Java classes, and vice versa.

23.1. Usage

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

Example 23.1. Using the Groovy plugin

build.gradle

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

23.2. Tasks

The Groovy plugin adds the following tasks to the project.

Table 23.1. Groovy plugin - tasks

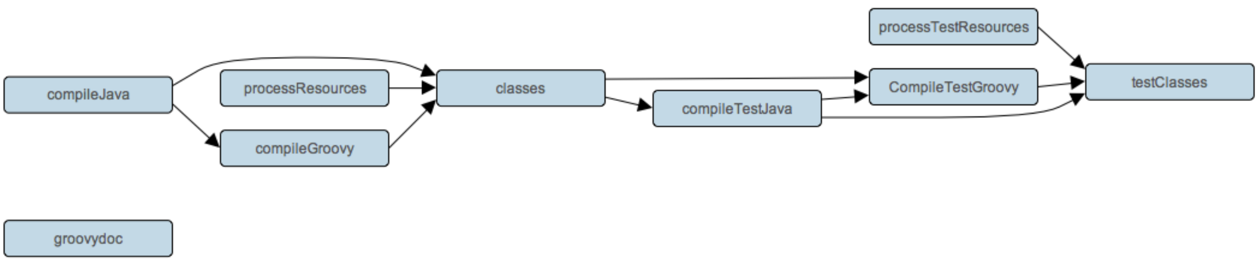
Task name	Depends on	Type	Description
`compileGroovy`	`compileJava`	`GroovyCompile`	Compiles production Groovy source files using groovyc.
`compileTestGroovy`	`compileTestJava`	`GroovyCompile`	Compiles test Groovy source files using groovyc.
`compile`*`SourceSet`*`Groovy`	`compile`*`SourceSet`*`Java`	`GroovyCompile`	Compiles the given source set's Groovy source files using groovyc.
`groovydoc`	-	`Groovydoc`	Generates API documentation for the production Groovy source files using groovydoc.

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

Table 23.2. Groovy plugin - additional task dependencies

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

Figure 23.1. Groovy plugin - tasks



23.3. Project layout

The Groovy plugin assumes the project layout shown in Table 23.3, “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. [10] None of these directories need exist or have anything in them. The Groovy plugin will compile whatever it finds, and handles anything which is missing.

Table 23.3. Groovy plugin - project layout

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

23.3.1. Changing the project layout

TBD

Example 23.2. Custom Groovy source layout

`build.gradle`

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

23.4. Dependency management

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

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

Example 23.3. Configuration of Groovy plugin

build.gradle

```
repositories {
    mavenCentral()
}

dependencies {
    groovy group: 'org.codehaus.groovy', name: 'groovy', version: '1.7.10'
}
```

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

Example 23.4. Configuration of Groovy plugin

build.gradle

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

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

23.5. Convention properties

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

23.6. Source set properties

The Groovy plugin adds the following convention properties to each source set in the project. You can use these properties in your build script as though they were properties of the source set object (see Section 20.2, "Using the convention object").

Table 23.4. 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 17.5, “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 23.5. 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.

23.7. CompileGroovy

The Groovy plugin adds a GroovyCompile instance for each source set in the project. The task type extends the Compile task (see Section 22.11, “CompileJava”). The compile task delegates to the Ant Groovyc task to do the compile. Via the compile task you can set most of the properties of Ants Groovyc task.

Table 23.6. Groovy plugin - CompileGroovy properties

Task Property	Type	Default Value
`classpath`	`FileCollection`	*`sourceSet`*`.compileCl`
`source`	`FileTree`. Can set using anything described in Section 17.5, “Specifying a set of input files”.	*`sourceSet`*`.groovy`
`destinationDir`	`File`.	*`sourceSet`*`.classesDi`
`groovyClasspath`	`FileCollection`	`groovy` configuration

---

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

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

24

The Scala Plugin

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

24.1. Usage

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

Example 24.1. Using the Scala plugin

build.gradle

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

24.2. Tasks

The Scala plugin adds the following tasks to the project.

Table 24.1. Scala plugin - tasks

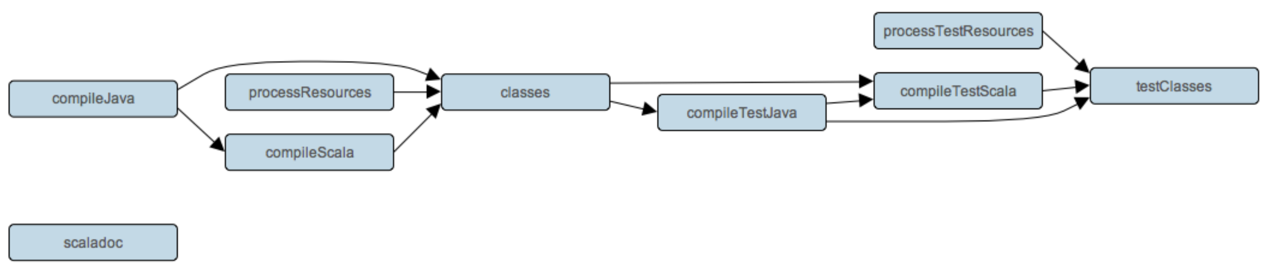
Task name	Depends on	Type	Description
`compileScala`	`compileJava`	`ScalaCompile`	Compiles production Scala source files using scalac.
`compileTestScala`	`compileTestJava`	`ScalaCompile`	Compiles test Scala source files using scalac.
compile*SourceSet*Scala	compile*SourceSet*Java	`ScalaCompile`	Compiles the given source set's Scala source files using scalac.
`scaladoc`	-	`ScalaDoc`	Generates API documentation for the production Scala source files using scaladoc.

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

Table 24.2. Scala plugin - additional task dependencies

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

Figure 24.1. Scala plugin - tasks



24.3. 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 exist or have anything in them. The Scala plugin will compile whatever it finds, and handles anything which is missing.

Table 24.3. Scala plugin - project layout

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

24.3.1. Changing the project layout

TBD

Example 24.2. Custom Scala source layout

`build.gradle`

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

24.4. Dependency Management

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

Example 24.3. Declaring the Scala version to use

build.gradle

```
repositories {
    mavenCentral()
}

dependencies {
    // Libraries needed to run the scala tools
    scalaTools 'org.scala-lang:scala-compiler:2.8.1'
    scalaTools 'org.scala-lang:scala-library:2.8.1'

    // Libraries needed for scala api
    compile 'org.scala-lang:scala-library:2.8.1'
}
```

24.5. Convention Properties

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

24.6. Source set properties

The Scala plugin adds the following convention properties to each source set in the project. You can use these properties in your build script as though they were properties of the source set object (see Section 20.2, “Using the convention object”).

Table 24.4. 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 .scal 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 17.5, “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 24.5. 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.

24.7. 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 24.4. 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

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.

25.1. Usage

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

Example 25.1. Using the War plugin

`build.gradle`

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

25.2. Tasks

The War plugin adds the following tasks to the project.

Table 25.1. 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 25.2. War plugin - additional task dependencies

Task name	Depends on
assemble	war

Figure 25.1. War plugin - tasks

25.3. Project layout

Table 25.3. War plugin - project layout

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

25.4. Dependency management

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

25.5. Convention properties

Table 25.4. 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.

25.6. 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 again may contain a `web.xml` file. Your compiled classes are compiled to `WEB-INF/classes`. All the dependencies of the `runtime` [11] configuration are copied to `WEB-INF/lib`.

Have also a look at `War`.

25.7. Customizing

Here is an example with the most important customization options:

Example 25.2. Customization of war plugin

`build.gradle`

```
```

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

---

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

26

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.

26.1. Usage

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

Example 26.1. Using the Ear plugin

build.gradle

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

26.2. Tasks

The Ear plugin adds the following tasks to the project.

Table 26.1. 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 26.2. Ear plugin - additional task dependencies

Task name	Depends on
assemble	ear

26.3. Project layout

Table 26.3. Ear plugin - project layout

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

26.4. 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.

26.5. Convention properties

Table 26.4. Ear plugin - directory properties

Property name	Type
`appDirName`	`String`
`libDirName`	`String`
`deploymentDescriptor`	`org.gradle.plugins.ear.descriptor.DeploymentDescr`

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

26.6. 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.

Also have a look at `Ear`.

26.7. Customizing

Here is an example with the most important customization options:

Example 26.2. Customization of ear plugin

`build.gradle`

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

repositories { mavenCentral() }

dependencies {
    //following dependencies will become the ear modules and placed in the ear
    deploy project(':war')

    //following dependencies will become ear libs and placed in a dir configure
    earlib group: 'log4j', name: 'log4j', version: '1.2.15', ext: 'jar'
}

ear {
    appDirName 'src/main/app'  // use application metadata found in this folde
    libDirName 'APP-INF/lib'  // put dependency libraries into APP-INF/lib insi
                                // also modify the generated deployment descrip
    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
        description = "My customized EAR for the Gradle documentation"  // defa
//      libraryDirectory = "APP-INF/lib"  // not needed, because setting libDir
//      module("my.jar", "java")  // wouldn't deploy since my.jar isn't a deplo
//      webModule("my.war", "/")  // wouldn't deploy since my.war isn't a deplo
        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`.

26.8. Using custom descriptor file

Let's say you already have the `application.xml` and want to use it instead of configuring the ea section. To accommodate that place the `META-INF/application.xml` in the right place inside your source folders (see the `appDirName` property). The existing file contents will be used and the explicit configuration in the `ear.deploymentDescriptor` will be ignored.

27

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.

27.1. Usage

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

Example 27.1. Using the Jetty plugin

build.gradle

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

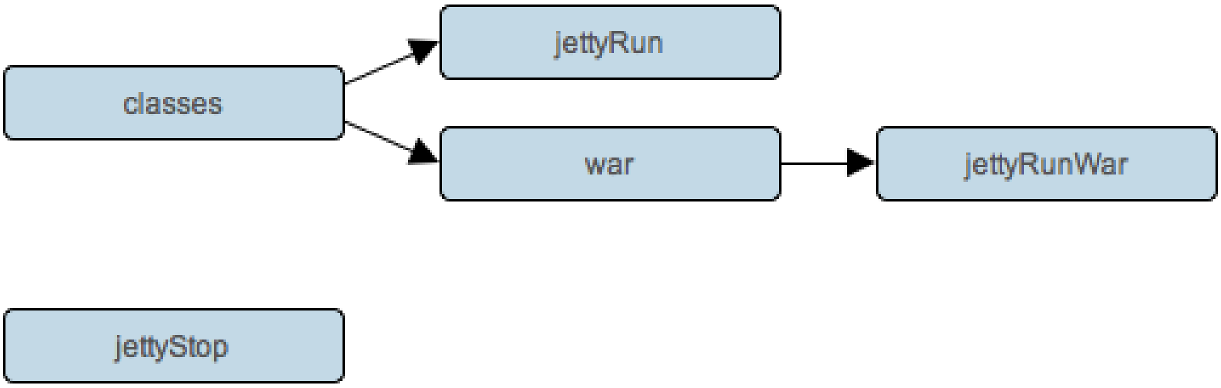
27.2. Tasks

The Jetty plugin defines the following tasks:

Table 27.1. 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 27.1. Jetty plugin - tasks



27.3. Project layout

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

27.4. Dependency management

The Jetty plugin does not define any dependency configurations.

27.5. Convention properties

The Jetty plugin defines the following convention properties:

Table 27.2. Jetty plugin - properties

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

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

28

The Checkstyle Plugin

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

28.1. Usage

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

Example 28.1. 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`**.

28.2. Tasks

The Checkstyle plugin adds the following tasks to the project:

Table 28.1. 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 28.2. Checkstyle plugin - additional task dependencies

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

28.3. Project layout

The Checkstyle plugin expects the following project layout:

Table 28.3. Checkstyle plugin - project layout

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

28.4. Dependency management

The Checkstyle plugin adds the following dependency configurations:

Table 28.4. Checkstyle plugin - dependency configurations

Name	Meaning
`checkstyle`	The Checkstyle libraries to use

28.5. Configuration

See `CheckstyleExtension`.

29

The CodeNarc Plugin

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

29.1. Usage

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

Example 29.1. 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`**.

29.2. Tasks

The CodeNarc plugin adds the following tasks to the project:

Table 29.1. 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 29.2. CodeNarc plugin - additional task dependencies

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

29.3. Project layout

The CodeNarc plugin expects the following project layout:

Table 29.3. CodeNarc plugin - project layout

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

29.4. Dependency management

The CodeNarc plugin adds the following dependency configurations:

Table 29.4. CodeNarc plugin - dependency configurations

Name	Meaning
`codenarc`	The CodeNarc libraries to use

29.5. Configuration

See `CodeNarcExtension`.

30

The FindBugs Plugin

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

30.1. Usage

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

Example 30.1. 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`**.

30.2. Tasks

The FindBugs plugin adds the following tasks to the project:

Table 30.1. 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 30.2. FindBugs plugin - additional task dependencies

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

30.3. Dependency management

The FindBugs plugin adds the following dependency configurations:

Table 30.3. FindBugs plugin - dependency configurations

Name	Meaning
`findbugs`	The FindBugs libraries to use

30.4. Configuration

See `FindBugsExtension`.

31

The JDepend Plugin

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

31.1. Usage

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

Example 31.1. 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`**.

31.2. Tasks

The JDepend plugin adds the following tasks to the project:

Table 31.1. 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 31.2. JDepend plugin - additional task dependencies

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

31.3. Dependency management

The JDepend plugin adds the following dependency configurations:

Table 31.3. JDepend plugin - dependency configurations

Name	Meaning
`jdepend`	The JDepend libraries to use

31.4. Configuration

See `JDependExtension`.

32

The PMD Plugin

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

32.1. Usage

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

Example 32.1. 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`**.

32.2. Tasks

The PMD plugin adds the following tasks to the project:

Table 32.1. 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 32.2. PMD plugin - additional task dependencies

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

32.3. Dependency management

The PMD plugin adds the following dependency configurations:

Table 32.3. PMD plugin - dependency configurations

Name	Meaning
`pmd`	The PMD libraries to use

32.4. Configuration

See `PmdExtension`.

33

The Sonar Plugin

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 and its subprojects. The results are stored in the Sonar database. The plugin requires Sonar 2.9 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.

33.1. Usage

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

Example 33.1. Applying the Sonar plugin

`build.gradle`

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

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

Example 33.2. 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"
    }
}
```

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 33.3. 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.

33.2. 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 and less likely to run into out-of-memory problems 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 33.4. 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 33.5. 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 33.6. 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 33.7. 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 33.8. Using property syntax

`build.gradle`

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

33.3. 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 33.9. Analyzing custom source sets

`build.gradle`

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

33.4. 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 33.10. Setting custom global properties

build.gradle

```
sonar.withGlobalProperties { props ->
    props["some.global.property"] = "some value"
}
```

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

Example 33.11. Setting custom project properties

build.gradle

```
sonar.project.withProjectProperties { props ->
    props["some.project.property"] = "some value"
}
```

The Sonar documentation provides a complete list of Sonar properties. (Note that most of these properties are already covered by the plugin's object model.) For configuring third-party Sonar plugins, consult their documentation.

33.5. Tasks

The Sonar plugin adds the following tasks to the project.

Table 33.1. Sonar plugin - tasks

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

34

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 22.13.1, “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.

34.1. Usage

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

Example 34.1. Using the OSGi plugin

build.gradle

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

34.2. Implicitly applied plugins

Applies the Java base plugin.

34.3. Tasks

This plugin does not add any tasks.

34.4. Dependency management

TBD

34.5. Convention object

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

34.5.1. Convention properties

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

34.5.2. Convention methods

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

Table 34.1. 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 there package dependencies and the packages they expose. Based on this the *Import-Package* and the *Export-Package* values of the OSGi Manifest are calculated. If the classpath contains jars with an OSGi bundle, the bundle information is used to specify version information for the *Import-Package* value. Beside the explicit properties of the `OsgiManifest` object you can add instructions.

Example 34.2. 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 Framew
        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. They are joined by Gradle with the `,` separator. To learn more about the available instructions have a look at the BND tool.

35

The Eclipse Plugin

The Eclipse plugin generates 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 1.0-milestone-4 WTP-generating code was refactored into a separate plugin called `eclipse` . So if you are interested in WTP integration then only apply the `eclipse-wtp` plugin. Otherwise applying `eclipse` plugin is enough. This change was requested by Eclipse users who take advantage of `war` or `ear` plugin but they don't use Eclipse WTP. Internally, `eclipse-wtp` 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 35.1. 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` file.
War	Adds web application support to `.project` file. Generates WTP settings files only if `eclipse-wtp` plugin was applied.
Ear	Adds ear application support to `.project` file. Generates WTP settings files only if ec plugin was applied.

The Eclipse plugin is open to customization and provides a standardized set of hooks for adding and removing content from the generated files.

35.1. Usage

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

Example 35.1. Using the Eclipse plugin

build.gradle

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

The Eclipse plugin adds a number of tasks to your projects. The main tasks that you will use are the `eclipse` and `cleanEclipse` tasks.

35.2. Tasks

The Eclipse plugin adds the tasks shown below to a project.

Table 35.2. Eclipse plugin - tasks

Task name	Depends on	Type
`eclipse`	`eclipseProject`, `eclipseClasspath`, `eclipseJdt`, `eclipseWtpComponent`, `cleanEclipseWtpFacet`	`Task`
`cleanEclipse`	`cleanEclipseProject`, `cleanEclipseClasspath`, `cleanEclipseJdt`, `cleanEclipseWtpComponent`, `cleanEclipseWtpFacet`	`Delete`
`cleanEclipseProject`	-	`Delete`
`cleanEclipseClasspath`	-	`Delete`
`cleanEclipseJdt`	-	`Delete`
`cleanEclipseWtpComponent`	-	`Delete`
`cleanEclipseWtpFacet`	-	`Delete`
`eclipseProject`	-	`GenerateEclipsePro`
`eclipseClasspath`	-	`GenerateEclipseCla`
`eclipseJdt`	-	`GenerateEclipseJdt`
`eclipseWtpComponent`	-	`GenerateEclipseWtp`
`eclipseWtpFacet`	-	`GenerateEclipseWtp`

35.3. Configuration

Table 35.3. Configuration of the Eclipse plugin

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 `ecli` plugin was applied.
`EclipseWtpFacet`	`eclipse.wtp.facet`	Allows configuring wtp facet information only if `eclipse-wtp` plugin was applied.

35.4. Customizing the generated files

The Eclipse plugin allows you to customise the generated metadata files. The plugin provides a DSL for configurinh 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 and configuration that the Eclipse plugin does not model.

35.4.1. 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.

35.4.1.1. Disabling merging with a complete overwrite

To completely overwrite existing Eclipse files, execute a clean task together with its corresponding generation task, for example **`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.

Complete overwrite works equally well for individual files, for example by executing **`gradle clear`**
.

35.4.2. Hooking into the generation lifecycle

The Eclipse plugin provides 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 35.4. Advanced configuration hooks

Model	**`beforeMerged { arg -> }`** argument type	**`whenMerged { arg -> }`** argument type	w a
`EclipseProject`	`Project`	`Project`	X
`EclipseClasspath`	`Classpath`	`Classpath`	X
`EclipseJdt`	`Jdt`	`Jdt`	
`EclipseWtpComponent`	`WtpComponent`	`WtpComponent`	X
`EclipseWtpFacet`	`WtpFacet`	`WtpFacet`	X

35.4.2.1. 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 35.2. Partial Overwrite for Classpath

`build.gradle`

```
eclipse.classpath.file {
    beforeMerged { classpath ->
        classpath.entries.removeAll { entry -> entry.kind == 'lib' || entry.ki
    }
}
```

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 35.3. Partial Overwrite for Project

`build.gradle`

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

35.4.2.2. 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 35.4. Export Dependencies

`build.gradle`

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

35.4.2.3. 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 35.5. 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.ja
}
```

36

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 36.1. 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.

36.1. Usage

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

Example 36.1. 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 `id` and `cleanIdea` tasks.

36.2. Tasks

The IDEA plugin adds the tasks shown below to a project. Notice that `clean` does not depend on c . It's because workspace contains a lot of user specific temporary data and typically it is not desirable to manipulate it outside IDEA.

Table 36.2. IDEA plugin - Tasks

Task name	Depends on	Type	Descriptior
`idea`	`ideaProject`, `ideaModule`, `ideaWorkspace`	-	Generates all IDEA configuratio files
`cleanIdea`	`cleanIdeaProject`, `cleanIdeaModule`	`Delete`	Removes al IDEA configuratio 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` fil
`ideaWorkspace`	-	`GenerateIdeaWorkspace`	Generates the `.iws` file. This task is only added to the root project.

36.3. Configuration

Table 36.3. 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

36.4. Customizing the generated files

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.

36.4.1. 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.

36.4.1.1. Disabling merging with a complete overwrite

To completely overwrite existing IDEA files, execute a clean task together with its corresponding generation task, for example **`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.

Complete overwrite works equally well for individual files, for example by executing **`gradle clear`** .

36.4.2. 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 36.4. Idea plugin hooks

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

36.4.2.1. Partial overwrite of existing content

A complete overwrite causes all existing content to be discarded, thereby losing any changes made directly in the IDE. 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 36.2. Partial Overwrite 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 36.3. Partial Overwrite for Project

`build.gradle`

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

36.4.2.2. 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 IDEA files. Here is how you would export all the dependencies of an IDEA module:

Example 36.4. Export Dependencies

`build.gradle`

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

36.4.2.3. 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 36.5. Customizing the XML

`build.gradle`

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

36.5. Further things to consider

The paths of the 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. If you use such a path variable, you need to configure this path variable via `idea.pathVariables`, so that it can do a proper merge without creating duplicates.

37

The Antlr Plugin

The Antlr plugin extends the Java plugin to add support for generating parsers using Antlr.

37.1. Usage

To use the Antlr plugin, include in your build script:

Example 37.1. Using the Antlr plugin

`build.gradle`

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

37.2. Tasks

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

Table 37.1. 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 37.2. Antlr plugin - additional task dependencies

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

37.3. Project layout

Table 37.3. 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.

37.4. Dependency management

The Antlr plugin adds an `antlr` dependency configuration. You use this to declare the version of Antlr you wish to use.

Example 37.2. Declare Antlr version

`build.gradle`

```
repositories {
    mavenCentral()
}

dependencies {
    antlr 'antlr:antlr:2.7.7'
}
```

37.5. Convention properties

The Antlr plugin does not add any convention properties.

37.6. Source set properties

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

Table 37.4. 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 found in the Antlr source directories, and excludes all other types of files.
antlr.srcDirs	Set<File>. Can set using anything described in Section 17.5, “Specifying a set of input files”.	[*projectDir*/src/*name*/antlr]	The source directories containing the Antlr grammar files of this source set.

38

The Project Report Plugin

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

The Project report plugin adds some tasks to your project which generate reports containing useful information about your build. Those tasks generate exactly the same content as the command line reports triggered by **`gradle tasks`**, **`gradle dependencies`** and **`gradle properties`** (see Section 11.5, “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.

38.1. Usage

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

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

38.2. Tasks

The project report plugin defines the following tasks:

Table 38.1. Project report plugin - tasks

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

38.3. Project layout

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

38.4. Dependency management

The project report plugin does not define any dependency configurations.

38.5. Convention properties

The project report defines the following convention properties:

Table 38.2. Project report plugin - convention properties

Property name	Type	Default value
`reportsDirName`	`String`	`reports`
`reportsDir`	`File` (read-only)	*`buildDir/reportsDirName`*
`projects`	`Set<Project>`	`A one element set with the project`
`projectReportDirName`	`String`	`project`
`projectReportDir`	`File` (read-only)	*`reportsDir/projectReportDirName`*

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

39

The Announce Plugin

The Gradle announce plugin enables you to publish messages on succeeded tasks to your favourite platforms. It supports

- Twitter
- Ubuntu Notify
- Snarl, a Windows Notification System
- Growl, a Mac OS X Notification System

39.1. Usage

To use the announce plugin, include in your build script:

Example 39.1. Using the announce plugin

`build.gradle`

```
apply plugin: 'announce'
```

After that, configure you username and password (if required for the service you want to announce to) with:

Example 39.2. Configure the announce plugin

`build.gradle`

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

Finally, you can use announce with any task by attaching it via task.dolast() as shown below

Example 39.3. Using the announce plugin

build.gradle

```
task helloWorld << {
    ant.echo(message: "hello")
}

helloWorld.doLast {
    announce("Build complete", "notify-send")
    announce("Build complete", "twitter")
}
```

As you can see, the syntax in `.doLast` is

```
announce("MESSAGE", "TARGET")
```

Where MESSAGE is any GString you pass (and might have constructed before). And TARGET might one of the following:

Table 39.1. announce plugin targets

target literal	target	configuration parameters	more information
twitter	Twitter	username , password	
snarl	Snarl Windows Notification Service		
growl	Growl Mac OS X Notification Service		
notify-send	Notify Ubuntu Notification Service		You need to have notify-send installed for this. Run `sudo apt-get install libnoti` on Ubuntu to install it.

39.2. Tasks

TBD

39.3. Project layout

TBD

39.4. Dependency management

TBD

39.5. Convention properties

The announce plugin adds an TBD

TBD

40

The Build Announcements Plugin

> The build announcements is experimental and should not be considered stable.

The build announcements plugin uses the announce plugin to send local announcements on important events in the build.

40.1. Usage

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

Example 40.1. 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 40.2. Using the build announcements plugin from an init script

init.gradle

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

41

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.

41.1. Usage

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

Example 41.1. 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 41.2. 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.

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 installApp`** to create an image of the application in `build/install/`*`projectName`*. You can run **`gradle dist`** to create a ZIP containing the distribution.

41.2. Tasks

The Application plugin adds the following tasks to the project.

Table 41.1. 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.
`installApp`	`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.

41.3. Convention properties

The application plugin adds some properties to the project, which you can use to configure its behaviour. See `Project`.

41.4. 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 `installApp` and `distZip` tasks as the specification of what is to be include in the distribution. Above copying the starting 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 41.3. 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 16.8.1, “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 41.4. 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
```

42

Dependency Management

42.1. Introduction

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

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

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

42.2. Dependency management overview

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

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

- Conflicting versions of the same jar should be detected and either resolved or cause an exception.

42.2.1. Versioning the jar name

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

42.2.2. Transitive dependency management

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

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

42.2.3. Version conflicts

In your dependency description you tell Gradle which version of a dependency is needed by another dependency. This frequently leads to conflicts. Different dependencies rely on different versions of another dependency. The JVM unfortunately does not offer yet any easy way, to have different versions of the same jar in the classpath (see Section 42.2.5, “Dependency management and Java”).

Gradle offers following conflict resolution strategies:

- *Newest* - used by default by Gradle - the newest version of the dependency is used. This strategy has been in Gradle since early days.

- *Fail* - fail eagerly on version conflict. Useful if you need extra control and manage the conflicts manually. Introduced in `1.0-milestone-6`. See `ResolutionStrategy` for reference on managing the conflict resolution strategies.

Gradle provides means to resolve version conflicts:

- Configuring a first level dependency as *forced*. The feature has been in Gradle since early days. This approach is useful if the dependency incurring conflict is already a first level dependency. See examples in `DependencyHandler`
- Configuring any dependency (transitive or not) as *forced*. The feature was introduced in `1.0-milestone-7`. This approach is useful if the dependency incurring conflict is a transitive dependency. It also can be used to force versions of first level dependencies. See examples in `ResolutionStrategy`

To deal with problems due to version conflicts, reports with dependency graphs are also very helpful. Such reports are another feature of dependency management.

42.2.4. Dynamic Versions and Changing Modules

Sometimes, you always want to use the latest version of a particular dependency, or the latest in a range of versions. You can easily do this using a *dynamic version*. A dynamic version can be either a version range (eg. `2.+`) or it can be a placeholder for the latest version available (eg. `latest.in`).

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 artifacts published.

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 42.7.3, “Fine-tuned control over dependency caching”).

42.2.5. Dependency management and Java

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

Both approaches rely on descriptor XML files, which contains information about the dependencies of a particular jar. Both also use repositories where the actual jars are placed together with their descriptor files. And both offer resolution for conflicting jar versions in one form or the other. Yet we think the differences of both approaches are significant in terms of flexibility and maintainability. Beside this, Ivy fully supports the Maven dependency handling. So with Ivy you have access to both worlds. We like Ivy very much. Gradle uses it under the hood for its dependency management. Ivy is most often used via Ant and XML descriptors. But it also has an API. We integrate deeply with Ivy via its API. This enables us to build new concepts on top of Ivy which Ivy does not offer itself.

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

42.3. 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 plugin add pre-defined configurations to your project. The Java plugin, for example, adds some configurations to represent the various classpaths it needs. see Section 22.5, “Dependency management” for details. Of course you can add your add custom configurations on top of that. There are many use cases for custom configurations. This is very handy for example for adding dependencies not needed for building or testing your software (e.g. additional JDBC drivers to be shipped with your distribution).

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 42.1. Definition of a configuration

`build.gradle`

```
configurations {
    compile
}
```

To access a configuration:

Example 42.2. Accessing a configuration

`build.gradle`

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

To configure a configuration:

Example 42.3. Configuration of a configuration

build.gradle

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

42.4. How to declare your dependencies

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

Table 42.1. 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.

42.4.1. External module dependencies

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

Example 42.4. 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
    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', trans
    runtime(group: 'org.hibernate', name: 'hibernate', version: '3.0.5') {
        transitive = true
    }
}
```

Please see the `DependencyHandler` for more examples and complete reference. Please read on to get thorough understanding of the Gradle's dependency management.

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

If you declare a module dependency, Gradle looks for a corresponding module descriptor file (`pom.` 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-3.0.5.jar`) as well as its dependencies (e.g. cglib) are downloaded. If no such module descriptor file exists, Gradle looks for a file called `hibernate-3.0` to retrieve. In Maven a module can only have one and only one artifact. In Gradle and Ivy a module can have multiple artifacts. Each artifact can have a different set of dependencies.

42.4.1.1. Depending on modules with multiple artifacts

As mentioned earlier, a maven module has only one artifact. So, when your project depends on a maven module it's obvious what artifact is the actual dependency. With Gradle or Ivy the case is different. Ivy model of dependencies (`ivy.xml`) can declare multiple artifacts. For more information, see Ivy reference for `ivy.xml`. In Gradle, when you declare a dependency on an ivy module you actually declare dependency on the `'default'` configuration of that module. So the actual list of artifacts (typically jars) your project depends on, are all artifacts that are attached to the `default` configuration of that module. This is very important in following exemplary use cases:

- The `default` configuration of some module contains some artifacts you don't want on the classpath. You might need to configure a dependency on specific artifact(s) of given module, rather than pulling all artifacts of the `default` dependency

- The artifact you need on the classpath has been published in a different configuration than the `default` one. This means this artifact will not be pulled in by Gradle. Unless you explicitly declare what configuration of the module you depend on.

There are other situations where it is necessary to fine-tune the dependency declaration. Please see the `DependencyHandler` for examples and complete reference on declaring dependencies.

42.4.1.2. 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. [12] 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 42.5. Artifact only notation

`build.gradle`

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

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

42.4.1.3. Classifiers

The Maven dependency management has the notion of classifiers. [13] Gradle supports this. To retrieve classified dependencies from a maven repository you can write:

Example 42.6. Dependency with classifier

`build.gradle`

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

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

To use the external dependencies of a configuration:

Example 42.7. Usage of external dependency of a configuration

`build.gradle`

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

Output of **`gradle -q listJars`**

```
> gradle -q listJars
hibernate-core-3.6.7.Final.jar
antlr-2.7.6.jar
commons-collections-3.1.jar
dom4j-1.6.1.jar
slf4j-api-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
```

42.4.2. Client module dependencies

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

Example 42.8. Client module dependencies - transitive dependencies

`build.gradle`

```
dependencies {
    runtime module("org.codehaus.groovy:groovy-all:1.8.4") {
        dependency("commons-cli:commons-cli:1.0") {
            transitive = false
        }
        module(group: 'org.apache.ant', name: 'ant', version: '1.8.2') {
            dependencies "org.apache.ant:ant-launcher:1.8.2@jar", "org.apache.a
        }
    }
}
```

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

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

42.4.3. 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 42.9. Project dependencies

build.gradle

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

For more information see the API documentation for `ProjectDependency`

Multi-project builds are discussed in Chapter 48, *Multi-project Builds*.

42.4.4. 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 42.10. 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 42.11. 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.na
}
```

Output of **`gradle -q list`**

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

42.4.5. 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 42.12. Gradle API dependencies

`build.gradle`

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

42.4.6. 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 42.13. Gradle's Groovy dependencies

`build.gradle`

```
dependencies {
    groovy localGroovy()
}
```

42.4.7. Excluding transitive dependencies

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

Example 42.14. 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. Have also a look at the API documentation of `Dependency` and `Configuration`.

42.4.8. Optional attributes

All attributes for a dependency are optional, except the name. It depends on the repository type, which information is need for actually finding the dependencies in the repository. See Section 42.6, “Repositories”. If you work for example with Maven repositories, you need to define the group, name and version. If you work with filesystem repositories you might only need the name or the name and the version.

Example 42.15. Optional attributes of dependencies

`build.gradle`

```
dependencies {
    runtime ":junit:4.8.2", ":testng"
    runtime name: 'testng'
}
```

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

Example 42.16. Collections and arrays of dependencies

build.gradle

```
List groovy = ["org.codehaus.groovy:groovy-all:1.8.4@jar",
               "commons-cli:commons-cli:1.0@jar",
               "org.apache.ant:ant:1.8.2@jar"]
List hibernate = ['org.hibernate:hibernate:3.0.5@jar', 'somegroup:someorg:1.0@j
dependencies {
    runtime groovy, hibernate
}
```

42.4.9. 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 available one anyway. If you work with Ivy repositories and want to declare a non-default configuration for your dependency you have to use the map notation and declare:

Example 42.17. Dependency configurations

build.gradle

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

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

Example 42.18. Dependency configurations for project

build.gradle

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

42.4.10. Dependency reports

You can generate dependency reports from the command line (see Section 11.5.3, “Listing project dependencies”). With the help of the Project report plugin (see Chapter 38, *The Project Report Plugin*) such a report can be created by your build.

42.5. Working with dependencies

For the examples below we have the following dependencies setup:

Example 42.19. 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:albatros:1.0"
}
```

The dependencies have the following transitive dependencies:

shark-1.0 -> seal-2.0, tuna-1.0

orca-1.0 -> seal-1.0

tuna-1.0 -> herring-1.0

You can use the configuration to access the declared dependencies or a subset of those:

Example 42.20. 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'
}
```

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

```
> gradle -q dependencies
albatros

albatros
orca
shark
tuna

albatros
shark
tuna
```

`dependencies` returns only the dependencies belonging explicitly to the configuration. `allDependencies` includes the dependencies from extended configurations.

To get the library files of the configuration dependencies you can do:

Example 42.21. 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
seal-2.0.jar
herring-1.0.jar
```

Sometimes you want the library files of a subset of the configuration dependencies (e.g. of a single dependency).

Example 42.22. 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 orginal 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 42.23. Configuration.copy

`build.gradle`

```
task copy << {
    configurations.alllife.copyRecursive { dep -> dep.name != 'orca' }.allDeper
        println dep.name
    }
    println()
    configurations.alllife.copy().allDependencies.each { dep ->
        println dep.name
    }
}
```

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

```
> gradle -q copy
albatros
shark
tuna

albatros
```

It is important to note that the returned files of the copied configuration are often but not always the same than the returned files of the dependency subset of the original configuration. In case of version conflicts between dependencies of the subset and dependencies not belonging to the subset the resolve result might be different.

Example 42.24. Configuration.copy vs. Configuration.files

`build.gradle`

```
task copyVsFiles << {
    configurations.sealife.copyRecursive { dep -> dep.name == 'orca' }.each {
        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`.

42.6. 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 42.2. Repository types

Type	Description
Maven central repository	A pre-configured repository that looks for dependencies in Maven Central.
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.

42.6.1. Maven central repository

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

Example 42.25. Adding central Maven repository

`build.gradle`

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

Now Gradle will look for your dependencies in this repository.

42.6.2. Local Maven repository

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

Example 42.26. Adding the local Maven cache as a repository

`build.gradle`

```
repositories {
    mavenLocal()
}
```

42.6.3. Maven repositories

For adding a custom Maven repository you can do:

Example 42.27. 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 42.28. 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.

42.6.3.1. 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 42.29. 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.

42.6.4. Flat directory repository

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

Example 42.30. 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 42.4.8, “Optional attributes”

42.6.5. Ivy repositories

To use an Ivy repository with a standard layout:

Example 42.31. Ivy repository

build.gradle

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

See `IvyArtifactRepository` for details.

42.6.5.1. Defining custom patterns for an Ivy repository

To define an Ivy repository with a non-standard layout, you can define a pattern layout for the repository:

Example 42.32. Ivy repository with pattern layout

build.gradle

```
repositories {
    ivy {
        url "http://repo.mycompany.com/repo"
        layout 'pattern', {
            artifact "[module]/[revision]/[artifact].[ext]"
            ivy "[module]/[revision]/ivy.xml"
        }
    }
}
```

42.6.5.2. Defining different artifact and ivy file locations for an Ivy repository

To define an Ivy repository which fetches ivy files and artifacts from different locations, you can explicitly define complete URL patterns for artifacts and ivy files:

Example 42.33. Ivy repository with custom patterns

build.gradle

```
repositories {
    ivy {
        artifactPattern "http://repo.mycompany.com/3rd-party-artifacts/[organis
        artifactPattern "http://repo.mycompany.com/company-artifacts/[organisa
        ivyPattern "http://repo.mycompany.com/ivy-files/[organisation]/[module]
    }
}
```

42.6.5.3. 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 42.34. Ivy repository

build.gradle

```
repositories {
    ivy {
        credentials {
            username 'user'
            password 'password'
        }
        artifactPattern "http://repo.mycompany.com/[organisation]/[module]/[rev
    }
}
```

42.6.6. Working with repositories

To access a repository:

Example 42.35. Accessing a repository

build.gradle

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

To configure a repository:

Example 42.36. Configuration of a repository

build.gradle

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

42.6.7. More about Ivy resolvers

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

- There are many options for the protocol to communicate with the repository (e.g. filesystem, http, ssh, ...)
- Each repository can have its own layout.

Let's say, you declare a dependency on the `junit:junit:3.8.2` library. Now how does Gradle find it in the repositories? Somehow the dependency information has to be mapped to a path. In contrast to Maven, where this path is fixed, with Gradle you can define a pattern that defines what the path will look like. Here are some examples: [14]

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

// Typical layout for an ivy repository (the organization is not split into sul
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 42.37. Definition of a custom repository

`build.gradle`

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

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.

42.7. 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 creating 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.

42.7.1. Key features of the Gradle dependency cache

42.7.1.1. 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 (eg 1.+) to a concrete version (eg 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.

42.7.1.2. 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 (see Section 42.7.1.3, “Artifact reuse”, below).

Dependency resolution will fail if the required artifacts are not available in any repository specified by the build, regardless whether the local cache has retrieved this artifact 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.

42.7.1.3. 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.

42.7.1.4. 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.

42.7.1.5. 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.

42.7.2. Command line options to override caching

42.7.2.1. 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.

42.7.2.2. 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 `--refre` 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 attempt to 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.

42.7.3. 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 42.38. 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 42.39. Changing module cache control

build.gradle

```
configurations.all {
    resolutionStrategy.cacheChangingModulesFor 30, 'days'
}
```

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

42.8. 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 has a very long response time.
- The `pom.xml`'s of many projects have wrong information (as one example, the POM of `comm` 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 wrong metadata in a Maven Central `pom.xml`.
- You don't want to expose people who want to build your project, to the downtimes or sometimes very long response times of Maven Central.

It is not a big deal to set-up a custom repository. [15] But it can be tedious, to keep it up to date. For a new version, you have always to create the new XML descriptor and the directories. And your custom repository is another infrastructure element which might have downtimes and needs to be updated. To enable historical builds, you need to keep all the past libraries and you need a backup. It is another layer of indirection. Another source of information you have to lookup. All this is not really a big deal but in its sum it has an impact. Repository Manager like Artifactory or Nexus make this easier. But for example open source projects don't usually have a host for those products.

This is a reason why some projects prefer to store their libraries in their version control system. This approach is fully supported by Gradle. The libraries can be stored in a flat directory without any XML module descriptor files. Yet Gradle offers complete transitive dependency management. You can use either client module dependencies to express the dependency relations, or artifact dependencies in case a first level dependency has no transitive dependencies. People can check out such a project from svn and have everything necessary to build it.

If you are working with a distributed version control system like Git you probably don't want to use the version control system to store libraries as people check out the whole history. But even here the flexibility of Gradle can make your life easier. For example you can use a shared flat directory without XML descriptors and yet you can have full transitive dependency management as described above.

You could also have a mixed strategy. If your main concern is bad metadata in the `pom.xml` and maintaining custom XML descriptors, *Client Modules* offer an alternative. But you can of course still use Maven2 repo and your custom repository as a repository for *jars only* and still enjoy *transitive* dependency management. Or you can only provide client modules for POMs with bad metadata. For the jars and the correct POMs you still use the remote repository.

42.8.1. Implicit transitive dependencies

There is another way to deal with transitive dependencies *without* XML descriptor files. You can do this with Gradle, but we don't recommend it. We mention it for the sake of completeness and comparison with other build tools.

The trick is to use only artifact dependencies and group them in lists. That way you have somehow expressed, what are your first level dependencies and what are transitive dependencies (see Section 42.4.8, "Optional attributes"). But the draw-back is, that for the Gradle dependency management all dependencies are considered first level dependencies. The dependency reports don't show your real dependency graph and the `compile` task uses all dependencies, not just the first level dependencies. All in all, your build is less maintainable and reliable than it could be when using client modules. And you don't gain anything.

---

[12] Gradle supports partial multiproject builds (seeChapter 48, *Multi-project Builds*).

[13] http://www.sonatype.com/books/maven-book/reference/pom-relationships-sect-project-relationships

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

[15] 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.

43

Publishing artifacts

43.1. 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.

43.2. 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`*`ConfigurationName`*. [16] Execution of these tasks will build or upload the artifacts belonging to the respective configuration.

Table Table 22.5, "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 43.5, "More about project libraries". As with dependencies, you can declare as many custom configurations as you like and assign artifacts to them.

43.3. Declaring artifacts

43.3.1. Archive task artifacts

You can use an archive task to define an artifact:

Example 43.1. 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.

43.3.2. File artifacts

You can also use a file to define an artifact:

Example 43.2. 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 43.3. 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 43.4. 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', builtI
}
```

43.4. Publishing artifacts

We have said that there is a specific upload task for each configuration. But before you can do an upload, you have to configure the upload task and define where to publish the artifacts to. The repositories you have defined (as described in Section 42.6, “Repositories”) are not automatically used for uploading. In fact, some of those repositories allow only for artifact downloading. Here is an example how you can configure the upload task of a configuration:

Example 43.5. 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 42.6.7, “More about Ivy resolvers”, you can use all the Ivy resolvers suitable for the purpose of uploading.

Uploading to a Maven repository is described in Section 44.6, “Interacting with Maven repositories”

.

43.5. 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 different group of artifacts which have a different set of dependencies. This mechanism is very powerful and flexible.

If someone wants to use your project as a library, she simply needs to declare on which configuration of the dependency to depend on. A Gradle dependency offers the `configuration` property to declare this. If this is not specified, the `default` configuration is used (see Section 42.4.9, “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 neccesary 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 44.6, “Interacting with Maven repositories”.

---

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

44

The Maven Plugin

This chapter is a work in progress

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

44.1. Usage

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

Example 44.1. Using the Maven plugin

build.gradle

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

44.2. Tasks

The Maven plugin defines the following tasks:

Table 44.1. 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 44.6.3, “Installing to the local repository”

44.3. Dependency management

The Maven plugin does not define any dependency configurations.

44.4. Convention properties

The Maven plugin defines the following convention properties:

Table 44.2. Maven plugin - properties

Property name	Type	Default value	Description
`pomDirName`	`String`	`poms`	The path of th directory to w the generated POMs, relativ to the build directory.
`pomDir`	`File (read-only)`	*`buildDir/pomDirName`*	The directory where the generated POMs are written to.
`conf2ScopeMappings`	`Conf2ScopeMappingContainer`	`n/a`	Instructions fo mapping Gra configurations Maven scope See Section 44.6. "Dependency mapping".

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

44.5. 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 44.2. 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`

44.6. Interacting with Maven repositories

44.6.1. 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.

44.6.2. 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 44.3. 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 some more work to do if you need support for other protocols than `file`. In this case the native Maven code we delegate to needs additional libraries. Which libraries depend on the protocol you need. The available protocols and the corresponding libraries are listed in Table 44.3, "Protocol jars for Maven deployment" (those libraries have again transitive dependencies which have transitive dependencies). [17] For example to use the ssh protocol you can do:

Example 44.4. Upload of file via SSH

`build.gradle`

```
configurations {
    deployerJars
}

repositories {
    mavenCentral()
}

dependencies {
    deployerJars "org.apache.maven.wagon:wagon-ssh:1.0-beta-2"
}

uploadArchives {
    repositories.mavenDeployer {
        name = 'sshDeployer' // optional
        configuration = configurations.deployerJars
        repository(url: "scp://repos.mycompany.com/releases") {
            authentication(userName: "me", password: "myPassword")
        }
    }
}
```

There are many configuration options for the Maven deployer. The configuration is done via a Groovy builder. All the elements of this tree are Java beans. To configure the simple attributes you pass a map to the bean elements. To add another bean elements to its parent, you use a closure. In the example above *repository* and *authentication* are such bean elements. Table 44.4, "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 44.3. Protocol jars for Maven deployment

Protocol	Library
http	org.apache.maven.wagon:wagon-http:1.0-beta-2
ssh	org.apache.maven.wagon:wagon-ssh:1.0-beta-2
ssh-external	org.apache.maven.wagon:wagon-ssh-external:1.0-beta-2
scp	org.apache.maven.wagon:wagon-scp:1.0-beta-2
ftp	org.apache.maven.wagon:wagon-ftp:1.0-beta-2
webdav	org.apache.maven.wagon:wagon-webdav-jackrabbit:1.0-beta-6
file	-

Table 44.4. 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

44.6.3. 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 `archives` 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.

44.6.4. Maven POM generation

The Maven POMs for uploading are automatically generated by Gradle. The groupId, artifactId, version and packaging values are taken from the project object. The dependency elements are created from the Gradle dependency declarations. You can find the generated POMs in the directory `<buildDir>/poms`. You can further customize the POM via the API of the `MavenPom` object.

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 44.5. Customization of pom

`build.gradle`

```
uploadArchives {
    repositories {
        mavenDeployer {
            repository(url: "file://localhost/tmp/myRepo/")
            pom.version = '1.0Maven'
            pom.artifactId = 'myMavenName'
        }
    }
}
```

Or you want to add new elements like license information.

Example 44.6. 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'
                    }
                }
            }
        }
    }
}
```

We use a builder here. You could also add the artifactId and groupId via the builder.

The pom object offers a `whenConfigure` method, if you need to modify the autogenerated content.

Example 44.7. Modifying auto-generated content

`build.gradle`

```
[installer, deployer]*.pom*.whenConfigured {pom ->
    pom.dependencies.find {dep -> dep.groupId == 'group3' && dep.artifactId ==
}
```

If you have more than one artifact to publish, things work a little bit differently. SeeSection 44.6.4.1, "Multiple artifacts per project".

To customize the settings for the Maven installer (seeSection 44.6.3, "Installing to the local repository”), you can do:

Example 44.8. Customization of Maven installer

`build.gradle`

```
configure(install.repositories.mavenInstaller) {
    pom.project {
        version '1.0Maven'
        artifactId 'myName'
    }
}
```

In contrast to the example above we use the builder here for changing groupId and artifactId.

44.6.4.1. 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 44.9. 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.

44.6.4.2. 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. One can assign also a priority to a particular configuration-to-scope mapping. Have a look at `Conf2ScopeMappingContainer` to learn more. To access the mapping configuration you can say:

Example 44.10. 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.

---

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

45

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).

45.1. Usage

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

Example 45.1. Using the Signing plugin

`build.gradle`

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

45.2. 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 property projects `signing.keyId`, `signing.password` and `signing.secretKeyRingFile` 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 envi
        Console console = System.console()
        console.printf "\n\nWe have to sign some things in this build.\n\nPleas
        allprojects*.setProperty("signing.keyId", console.readLine("PGP Key Id
        allprojects*.setProperty("signing.secretKeyRingFile", console.readLine(
        allprojects*.setProperty("signing.password", console.readPassword("PGP
        console.printf "\nThanks.\n\n"
    }
}
```

45.3. 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.

45.3.1. Signing Configurations

It is common to want to sign the artifacts of a configuration. For example, the Java plugin configures a jar to built 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 45.2. 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 `ar` artifacts (if needed) and then generate signatures for them. The signature files will be placed alongside the artifacts being signed.

Example 45.3. Signing a configuration output

Output of **`gradle signArchives`**

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

BUILD SUCCESSFUL

Total time: 1 secs
```

45.3.2. 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 45.4. 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 45.5. 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.

45.3.3. 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 45.6. Conditional signing

build.gradle

```
version = '1.0-SNAPSHOT'
isReleaseVersion = !version.endsWith("SNAPSHOT")

signing {
    required { isReleaseVersion && gradle.taskGraph.hasTask("uploadArchives") }
    sign configurations.archives
}
```

In this example, we only want to require signing if we are building a release version and we are going to publish it. Because we are inspecting the task graph to determine if we are going to be publishing, we must set the `signing.required` property to a closure to defer the evaluation. See `SigningExtension.setRequired()` for more information.

45.4. 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.

45.5. 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 45.7. Signing a POM for deployment

build.gradle

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

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.

46

C++ Support

> The Gradle C++ support is experimental and in very early stages of development so should not be considered stable.

The C++ plugins add support for building software comprised of C++ source code, and managing the process of building "native" software in general. While many excellent build tools exist for this space of software development, Gradle brings the dependency management practices more traditionally found in the JVM development space to C++ developers.

Currently, support is restricted to building with the g++ (i.e. the C++ focussed frontend to GCC) compiler/linker. Furthermore, there is no direct support for creating multiple variants of the same binary (e.g. 32 bit vs. 64 bit) and there is no direct support for cross platform source configuration (à la autoconf) at this time. Support for different compiler chains, managing multiple variants and cross platform source configuration will be added over time, making Gradle a fully capable build tool for C++ (and other "native" language) projects.

46.1. Usage

The build scripts DSLs, model elements and tasks used to manage C++ projects are added by the c plugin. However, it is typically more convenient to use either the `cpp-lib` or `cpp-exe` plugins that sit on top of the `cpp` plugin to preconfigure the project to build either a shared library or executable binary respectively.

Example 46.1. Using the 'cpp-exe' plugin

`build.gradle`

```
apply plugin: "cpp-exe"
```

Example 46.2. Using the 'cpp-lib' plugin

`build.gradle`

```
apply plugin: "cpp-lib"
```

The `cpp-exe` plugin configures the project to build a single executable (at `$buildDir/binaries`) and the `cpp-lib` plugin configures the project to build a single shared library (at `$buildDir/bi`).

46.2. Source code locations

Both plugins configure the project to look for `.cpp` source files in `src/main/cpp` and use the `src/` directory as a header include root. For a library, the header files in `src/main/headers` are considered the “public” or “exported” headers. Header files that should not be exported (but are used internally) 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).

46.3. Compiling

For both the `cpp-lib` and `cpp-exe` plugins, you can run the `compileMain` task.

As previously stated, the C++ support is currently based on the `g++` tool which must be installed and on the `PATH` for the Gradle process.

Arbitrary arguments can be provided to `g++` by using the following syntax:

Example 46.3. Supplying arbitrary args to gpp

`build.gradle`

```
executables {
    main {
        spec {
            args "-fno-access-control", "-fconserve-space"
        }
    }
}
```

The above example applies to the `cpp-exe` plugin, to supply arguments for the `cpp-lib` plugin replace “`executables`” with “`libraries`”.

46.4. Dependencies

Dependencies for C++ projects are binary libraries that export header files. The header files are used during compilation, with the compiled binary dependency being used during the linking.

46.4.1. External Dependencies

External dependencies (i.e. from a repository, not a subproject) must be specified using the following syntax:

Example 46.4. Declaring dependencies

`build.gradle`

```
cpp {
    sourceSets {
        main {
            dependency group: "some-org", name: "some-lib", version: "1.0"
        }
    }
}
```

Each dependency must be specified with the `dependency` method as above and must be declared as part of the source set. The `group`, `name` and `version` arguments *must* be supplied.

For each declared dependency, two actual dependencies are created. One with the classifer "`head`" and extension "`zip`" which is a zip file of the exported headers, and another with the classifier "`so`" and extension "`so`" which is the compiled library binary to link against (which is supplied as a direct input to the g++ link operation).

46.4.2. Project Dependencies

The notation for project dependencies is slightly different.

Example 46.5. Declaring project dependencies

`build.gradle`

```
project(":lib") {
    apply plugin: "cpp-lib"
}

project(":exe") {
    apply plugin: "cpp-exe"
    cpp {
        sourceSets {
            main {
                libs << project(":lib").libraries.main
            }
        }
    }
}
```

46.5. Publishing

The `cpp-exe` and `cpp-lib` plugins configure their respective output binaries to be publishable as part of the `archives` configuration. To publish, simply configure the `uploadArchives` task as per usual.

Example 46.6. Uploading exe or lib

`build.gradle`

```
group = "some-org"
archivesBaseName = "some-lib"
version = 1.0

uploadArchives {
    repositories {
        mavenDeployer {
            repository(url: uri("${buildDir}/repo"))
        }
    }
}
```

The `cpp-exe` plugin publishes a single artifact with extension “`exe`”. The `cpp-lib` plugin publishes two artifacts; one with classifier “`headers`” and extension “`zip`”, and one with classifier “`s`” and extension “`so`” (which is the format used when consuming dependencies).

> Currently, there is no support for publishing the dependencies of artifacts in POM or Ivy files. Future versions will support this.

47

The Build Lifecycle

We said earlier, that the core of Gradle is a language for dependency based programming. In Gradle terms this means that you can define tasks and dependencies between tasks. Gradle guarantees that these tasks are executed in the order of their dependencies, and that each task is executed only once. Those tasks form a Directed Acyclic Graph. There are build tools that build up such a dependency graph as they execute their tasks. Gradle builds the complete dependency graph *before* any task is executed. This lies at the heart of Gradle and makes many things possible which would not be possible otherwise.

Your build scripts configure this dependency graph. Therefore they are strictly speaking *build configuration scripts*.

47.1. Build phases

A Gradle build has three distinct phases.

Initialization
: Gradle supports single and multi-project builds. During the initialization phase, Gradle determines which projects are going to take part in the build, and creates a `Project` instance for each of these projects.

Configuration
: The build scripts of *all* projects which are part of the build are executed. This configures the project objects.

Execution
: Gradle determines the subset of the tasks, created and configured during the configuration phase, to be executed. The subset is determined by the task name arguments passed to the **gradle** command and the current directory. Gradle then executes each of the selected tasks.

47.2. Settings file

Beside the build script files, Gradle defines a settings file. The settings file is determined by Gradle via a naming convention. The default name for this file is `settings.gradle`. Later in this chapter we explain, how Gradle looks for a settings file.

The settings file gets executed during the initialization phase. A multiproject build must have a `sett` file in the root project of the multiproject hierarchy. It is required because in the settings file it is defined, which projects are taking part in the multi-project build (see Chapter 48, *Multi-project Builds*). For a single-project build, a settings file is optional. You might need it for example, to add libraries to your build script classpath (see Chapter 51, *Organizing Build Logic*). Let's first do some introspection with a single project build:

Example 47.1. Single project build

`settings.gradle`

```
println 'This is executed during the initialization phase.'
```

`build.gradle`

```
println 'This is executed during the configuration phase.'

task configured {
    println 'This is also executed during the configuration phase.'
}

task test << {
    println 'This is executed during the execution phase.'
}
```

Output of **`gradle test`**

```
> gradle test
This is executed during the initialization phase.
This is executed during the configuration phase.
This is also executed during the configuration phase.
:test
This is executed during the execution phase.

BUILD SUCCESSFUL

Total time: 1 secs
```

For a build script, the property access and method calls are delegated to a project object. Similarly property access and method calls within the settings file is delegated to a settings object. Have a look at `Settings`.

47.3. 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 48, *Multi-project Builds*).

47.3.1. Project locations

Multi-project builds are always represented by a tree with a single root. Each element in the tree represent a project. A project has a virtual and a physical path. The virtual path denotes the position of the project in the multi-project build tree. The project tree is created in the `settings.gr` 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.

47.3.2. 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.

47.3.2.1. Hierarchical layouts

Example 47.2. Hierarchical layout

`settings.gradle`

```
include 'project1', 'project2', 'project2:child1'
```

The `include` method takes as an argument a relative virtual path to the root project. This relative virtual path is assumed to be equal to the relative physical path of the subproject to the root project. You only need to specify the leafs of the tree. Each parent path of the leaf project is assumed to be another subproject which obeys to the physical path assumption described above.

47.3.2.2. Flat layouts

Example 47.3. Flat layout

`settings.gradle`

```
includeFlat 'project3', 'project4'
```

The `includeFlat` method takes directory names as an argument. Those directories need to exist at the same level as the root project directory. The location of those directories are considered as child projects of the root project in the virtual multi-project tree.

47.3.3. 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 47.4. 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 47.5. Modification of elements of the project tree

`settings.gradle`

```
rootProject.name = 'main'
project(':projectA').projectDir = new File(settingsDir, '../my-project-a')
project(':projectA').buildFileName = 'projectA.gradle'
```

Have a look at `ProjectDescriptor` for more details.

47.4. Initialization

How does Gradle know whether to do a single or multiproject build? If you trigger a multiproject build from the directory where the settings file is, things are easy. But Gradle also allows you to execute the build from within any subproject taking part in the build. [18] If you execute Gradle from within a project that has no `settings.gradle` file, Gradle does the following:

- It searches for a `settings.gradle` in a directory called `master` which has the same nesting level as the current dir.
- If no `settings.gradle` is found, it searches the parent directories for the existence of a `se` file.
- If no `settings.gradle` file is found, the build is executed as a single project build.
- If a `settings.gradle` file is found, Gradle checks if the current project is part of the multiproject hierarchy defined in the found `settings.gradle` file. If not, the build is executed as a single project build. Otherwise a multiproject build is executed.

What is the purpose of this behavior? Somehow Gradle has to find out, whether the project you are into, is a subproject of a multiproject build or not. Of course, if it is a subproject, only the subproject and its dependent projects are build. But Gradle needs to create the build configuration for the whole multiproject build (see Chapter 48, *Multi-project Builds*). Via the `-u` command line option, you can tell Gradle not to look in the parent hierarchy for a `settings.gradle` file. The current project is then always build as a single project build. If the current project contains a `settings.gr` file, the `-u` option has no meaning. Such a build is always executed as:

- a single project build, if the `settings.gradle` file does not define a multiproject hierarchy
- a multiproject build, if the `settings.gradle` file does define a multiproject hierarchy.

The auto search for a settings file does only work for multi-project builds with a physical hierarchical or flat layout. For a flat layout you must additionally obey to the naming convention described above. Gradle supports arbitrary physical layouts for a multiproject build. But for such arbitrary layouts you need to execute the build from the directory where the settings file is located. For how to run partial builds from the root see Section 48.4, “Running tasks by their absolute path”. In our next release we want to enable partial builds from subprojects by specifying the location of the settings file as a command line parameter. Gradle creates Project objects for every project taking part in the build. For a single project build this is only one project. For a multi-project build these are the projects specified in Settings object (plus the root project). Each project object has by default a name equals to the name of its top level directory. Every project except the root project has a parent project and might have child projects.

47.5. 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 48, *Multi-project Builds*.

47.6. Responding to the lifecycle in the build script

Your build script can receive notifications as the build progresses through its lifecyle. These notifications generally take 2 forms: You can either implement a particular listener interface, or you can provide a closure to execute when the notification is fired. The examples below use closures. For details on how to use the listener interfaces, refer to the API documentation.

47.6.1. Project evaluation

You can receive a notification immediately before and after a project is evaluated. This can be used to do things like performing additional configuration once all the definitions in a build script have been applied, or for some custom logging or profiling.

Below is an example which adds a `test` task to each project with the `hasTests` property set to true.

Example 47.6. 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 47.7. 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.

47.6.2. 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 47.8. Setting of certain property to all tasks

`build.gradle`

```
tasks.whenTaskAdded { task ->
    task.srcDir = 'src/main/java'
}

task a

println "source dir is $a.srcDir"
```

Output of **`gradle -q a`**

```
> gradle -q a
source dir is src/main/java
```

You can also add an `Action` to a `TaskContainer` to receive these events.

47.6.3. 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.

47.6.4. 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 47.9. 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.

---

[18] Gradle supports partial multiproject builds (see Chapter 48, *Multi-project Builds*).

48

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.

48.1. Cross project configuration

Let's start with a very simple multi-project build. After all Gradle is a general purpose build tool at its core, so the projects don't have to be java projects. Our first examples are about marine life.

48.1.1. Defining common behavior

We have the following project tree. This is a multi-project build with a root project `water` and a subproject `bluewhale`.

Example 48.1. 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/fir` which is in both the binary and source distributions 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 48.2. 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 48.3. 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/add` which is in both the binary and source distributions of Gradle.

`settings.gradle`

```
include 'bluewhale', 'krill'
```

Now we rewrite the `water` build script and boil it down to a single line.

Example 48.4. 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 `allproje` which returns a list with the current project and all its subprojects underneath it. If you call `allproj` with a closure, the statements of the closure are delegated to the projects associated with `allproj` . 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.

48.2. Subproject configuration

The Project API also provides a property for accessing the subprojects only.

48.2.1. Defining common behavior

Example 48.5. Defining common behaviour of all projects and subprojects

build.gradle

```
allprojects {
    task hello << {task -> println "I'm $task.project.name" }
}
subprojects {
    hello << {println "- I depend on water"}
}
```

Output of **gradle -q hello**

```
> gradle -q hello
I'm water
I'm bluewhale
- I depend on water
I'm krill
- I depend on water
```

48.2.2. 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 b project like this:

Example 48.6. 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 48.7. 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/spr` which is in both the binary and source distributions of Gradle.

`settings.gradle`

```
include 'bluewhale', 'krill'
```

`bluewhale/build.gradle`

```
hello.doLast { println "- I'm the largest animal that has ever lived on this p
```

`krill/build.gradle`

```
hello.doLast {
    println "- The weight of my species in summer is twice as heavy as all huma
}
```

`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.
```

48.2.3. Project filtering

To show more of the power of configuration injection, let's add another project called `tropicalFis` and add more behavior to the build via the build script of the `water` project.

48.2.3.1. Filtering by name

Example 48.8. 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/add` which is in both the binary and source distributions 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.

48.2.3.2. Filtering by properties

Using the project name for filtering is one option. Using dynamic project properties is another.

Example 48.9. 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/tro` which is in both the binary and source distributions of Gradle.

`settings.gradle`

```
include 'bluewhale', 'krill', 'tropicalFish'
```

`bluewhale/build.gradle`

```
arctic = true
hello.doLast { println "- I'm the largest animal that has ever lived on this p
```

`krill/build.gradle`

```
arctic = true
hello.doLast {
    println "- The weight of my species in summer is twice as heavy as all huma
}
```

`tropicalFish/build.gradle`

```
arctic = false
```

`build.gradle`

```
allprojects {
    task hello << {task -> println "I'm $task.project.name" }
}
subprojects {
    hello {
        doLast {println "- I depend on water"}
        afterEvaluate { Project project ->
            if (project.arctic) { doLast {
                println '- I love to spend time in the arctic waters.' }
            }
        }
    }
}
```

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

```
> gradle -q hello
I'm water
I'm bluewhale
- I depend on water
- I'm the largest animal that has ever lived on this planet.
- I love to spend time in the arctic waters.
I'm krill
- I depend on water
- The weight of my species in summer is twice as heavy as all human beings.
- I love to spend time in the arctic waters.
I'm tropicalFish
- I depend on water
```

In the build file of the `water` project we use an `afterEvaluate` notification. This means that the closure we are passing gets evaluated *after* the build scripts of the subproject are evaluated. As the property `arctic` is set in those build scripts, we have to do it this way. You will find more on this topic in Section 48.6, “Dependencies - Which dependencies?”

48.3. Execution rules for multi-project builds

When we have executed the `hello` task from the root project dir things behaved in an intuitive way. All the `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 48.10. 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` an 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 48.11. Evaluation and execution of projects

`bluewhale/build.gradle`

```
arctic = true
hello << { println "- I'm the largest animal that has ever lived on this plane

task distanceToIceberg << {
    println '20 nautical miles'
}
```

`krill/build.gradle`

```
arctic = true
hello << { println "- The weight of my species in summer is twice as heavy as a

task distanceToIceberg << {
    println '5 nautical miles'
}
```

Output of **`gradle -q distanceToIceberg`**

```
> gradle -q distanceToIceberg
20 nautical miles
5 nautical miles
```

Here the output without the `-q` option:

Example 48.12. 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!

48.4. 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 48.5, "Project and task paths"):

Example 48.13. 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 there absolute path, the last task is executed on the name matching mechanism described above.

48.5. Project and task paths

A project path has the following pattern: It starts always with a colon, which denotes the root project. The root project is the only project in a path that is not specified by its name. The path `:blu` corresponds to the file system path `water/bluewhale` in the case of the example above.

The path of a task is simply its project path plus the task name. For example `:bluewhale:hello`. Within a project you can address a task of the same project just by its name. This is interpreted as a relative path.

Originally Gradle has used the `'/'` character as a natural path separator. With the introduction of directory tasks (see Section 14.1, "Directory creation") this was no longer possible, as the name of the directory task contains the `'/'` character.

48.6. Dependencies - Which dependencies?

The examples from the last section were special, as the projects had no *Execution Dependencies*. They had only *Configuration Dependencies*. Here is an example where this is different:

48.6.1. Execution dependencies

48.6.1.1. Dependencies and execution order

Example 48.14. 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/dep` which is in both the binary and source distributions of Gradle.

`settings.gradle`

```
include 'consumer', 'producer'
```

`consumer/build.gradle`

```
task action << {
    println("Consuming message: " +
            (rootProject.hasProperty('producerMessage') ? rootProject.producerM
}
```

`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 did not work out. If nothing else is defined, Gradle executes the task in alphanumeric order. Therefore `:consumer:action` is executed before `:producer:action`. Let's try to solve this with a hack and rename the producer project to `aProducer`.

Example 48.15. 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.hasProperty('producerMessage') ? rootProject.producerM
}
```

Output of **`gradle -q action`**

```
> gradle -q action
Producing message:
Consuming message: Watch the order of execution.
```

Now we take the air out of this hack. We simply switch to the `consumer` dir and execute the build.

Example 48.16. Dependencies and execution order

Output of **`gradle -q action`**

```
> gradle -q action
Consuming message: null
```

For Gradle the two `action` tasks are just not related. If you execute the build from the `messages` project Gradle executes them both because they have the same name and they are down the hierarchy. In the last example only one `action` was down the hierarchy and therefore it was the only task that got executed. We need something better than this hack.

48.6.1.2. Declaring dependencies

Example 48.17. 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/dep` which is in both the binary and source distributions of Gradle.

`settings.gradle`

```
include 'consumer', 'producer'
```

`consumer/build.gradle`

```
dependsOn(':producer')

task action << {
    println("Consuming message: " +
            (rootProject.hasProperty('producerMessage') ? rootProject.producerI
}
```

`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 48.18. Declaring dependencies

Output of **`gradle -q action`**

```
> gradle -q action
Producing message:
Consuming message: Watch the order of execution.
```

We have now declared that the `consumer` project has an *execution dependency* on the `producer` project. For Gradle declaring *execution dependencies* between *projects* is syntactic sugar. Under the hood Gradle creates task dependencies out of them. You can also create cross project tasks dependencies manually by using the absolute path of the tasks.

48.6.1.3. The nature of project dependencies

Let's change the naming of our tasks and execute the build.

Example 48.19. Project execution dependencies

`consumer/build.gradle`

```
dependsOn(':producer')

task consume << {
    println("Consuming message: " +
            (rootProject.hasProperty('producerMessage') ? rootProject.producerM
}
```

`producer/build.gradle`

```
task produce << {
    println "Producing message:"
    rootProject.producerMessage = 'Watch the order of execution.'
}
```

Output of **`gradle -q consume`**

```
> gradle -q consume
Consuming message: null
```

Oops. Why does this not work? The `dependsOn` command is created for projects with a common lifecycle. Provided you have two Java projects where one depends on the other. If you trigger a compile for the dependent project you don't want that *all* tasks of the other project get executed. Therefore a `dependsOn` creates dependencies between tasks with equal names. To deal with the scenario above you would do the following:

Example 48.20. Cross project task dependencies

`consumer/build.gradle`

```
task consume(dependsOn: ':producer:produce') << {
    println("Consuming message: " +
            (rootProject.hasProperty('producerMessage') ? rootProject.producerM
}
```

`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.
```

48.6.2. Configuration time dependencies

Let's have one more example with our producer-consumer build before we enter *Java* land. We add a property to the producer project and create now a configuration time dependency from consumer on producer.

Example 48.21. Configuration time dependencies

`consumer/build.gradle`

```
message = rootProject.hasProperty('producerMessage') ? rootProject.producerMess

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 the projects is alphanumeric (for the same nesting level). Therefore the `consumer` project is evaluated before the `producer` project and the `key` value of the `produce` is set *after* it is read by the `consumer` project. Gradle offers a solution for this.

Example 48.22. Configuration time dependencies - evaluationDependsOn

`consumer/build.gradle`

```
evaluationDependsOn(':producer')

message = rootProject.hasProperty('producerMessage') ? rootProject.producerMes

task consume << {
    println("Consuming message: " + message)
}
```

Output of **`gradle -q consume`**

```
> gradle -q consume
Consuming message: Watch the order of evaluation.
```

The command `evaluationDependsOn` triggers the evaluation of `producer` *before* `consumer` is evaluated. The example is a bit contrived for the sake of showing the mechanism. In *this* case there would be an easier solution by reading the key property at execution time.

Example 48.23. Configuration time dependencies

`consumer/build.gradle`

```
task consume << {
    println("Consuming message: " +
            (rootProject.hasProperty('producerMessage') ? rootProject.producerI
}
```

Output of **`gradle -q consume`**

```
> gradle -q consume
Consuming message: Watch the order of evaluation.
```

Configuration dependencies are very different to execution dependencies. Configuration dependencies are between projects whereas execution dependencies are always resolved to task dependencies. Another difference is that always all projects are configured, even when you start the build from a subproject. The default configuration order is top down, which is usually what is needed.

On the same nesting level the configuration order depends on the alphanumeric position. The most common use case is to have multi-project builds that share a common lifecycle (e.g. all projects use the Java plugin). If you declare with `dependsOn` a *execution dependency* between different projects, the default behavior of this method is to create also a *configuration* dependency between the two projects. Therefore it is likely that you don't have to define configuration dependencies explicitly.

48.6.3. Real life examples

Gradle's multi-project features are driven by real life use cases. The first example for describing such a use case, consists of two webapplication projects and a parent project that creates a distribution out of them. [19] For the example we use only one build script and do *cross project configuration*.

Example 48.24. Dependencies - real life example - crossproject configuration

Build layout

```
webDist/
  settings.gradle
  build.gradle
  date/
    src/main/java/
      org/gradle/sample/
        DateServlet.java
  hello/
    src/main/java/
      org/gradle/sample/
        HelloServlet.java
```

Note: The code for this example can be found at `samples/userguide/multiproject/dep` which is in both the binary and source distributions of Gradle.

`settings.gradle`

```
include 'date', 'hello'
```

`build.gradle`

```
dependsOnChildren()

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 w . The *execution* dependency is in the other direction, as `webDist` depends on the build artifacts of d and `hello`. There is even a third dependency. `webDist` has a *configuration* dependency on `date` and `hello` because it needs to know the `archivePath`. But it asks for this information at *execution time*. Therefore we have no circular dependency.

Such and other dependency patterns are daily bread in the problem space of multi-project builds. If a build system does not support such patterns, you either can't solve your problem or you need to do ugly hacks which are hard to maintain and massively afflict your productivity as a build master.

There is one more thing to note from the current example. We have used the command `dependsOn` . It is a convenience method and calls the `dependsOn` method of the parent project for every child project (not every sub project). It declares a `execution` dependency of `webDist` on `date` and he .

Another use case would be a situation where the subprojects have a configuration *and* execution dependency on the parent project. This is the case when the parent project does configuration injection into its subprojects, and additionally produces something at execution time that is needed by its child projects (e.g. code generation). In this case the parent project would call the `childrenD` method to create an execution dependency for the child projects. We might add an example for this in a future version of the user guide.

48.7. Project lib dependencies

What if one projects needs the jar produced by another project in its compile path? And not just the jar but also the transitive dependencies of this jar? Obviously this is a very common use case for Java multi-project builds. As already mentioned in Section 42.4.3, “Project dependencies”, Gradle offers project lib dependencies for this.

Example 48.25. 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/dep` which is in both the binary and source distributions of Gradle.

We have the projects `shared`, `api` and`personService`. `personService` has a lib dependency on the other two projects. `api` has a lib dependency on `shared`. [20]

Example 48.26. 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:4.8.2"
    }
}

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

project(':services:personService') {
    dependencies {
        compile project(':shared'), project(':api')
    }
}

dependsOnChildren()
```

All the build logic is in the `build.gradle` of the root project. [21] 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 `shared` is built and then `api` 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 48.27. 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:4.8.2", project(':api')
    }
}
```

The Java plugin adds per default a jar to your project libraries which contains all the classes. In this example we create an *additional* library containing only the interfaces of the `api` project. We assign this library to a new *dependency configuration*. For the person service we declare that the project should be compiled only against the `api` interfaces but tested with all classes from `api`.

48.7.1. 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.

48.8. Multi-Project Building and Testing

The `build` task of the Java plugin is typically used to compile, test, and perform code style checks (if the CodeQuality plugin is used) of a single project. In multi-project builds you may often want to do all of these tasks across a range of projects. The `buildNeeded` and `buildDependents` tasks can help with this.

Let's use the project structure shown in Example 48.26, “Project lib dependencies”. In this example :services:personservice depends on both :api and :shared. The :api project also depends on

:shared.

Assume you are working on a single project, the :api project. You have been making changes, but have not built the entire project since performing a clean. You want to build any necessary supporting jars, but only perform code quality and unit tests on the project you have changed. The b task does this.

Example 48.28. 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 :shared:compile checking to see what has changed in the :shared project. Adding the `-a` option will cause gradle to use cached jars to resolve any project lib dependencies and not try to re-build the depended on projects.

Example 48.29. Partial Build and Test Single Project

Output of **`gradle -a :api:build`**

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

BUILD SUCCESSFUL

Total time: 1 secs
```

If you have just gotten the latest version of source from your version control system which included changes in other projects that :api depends on, you might want to not only build all the projects you depend on, but test them as well. The `buildNeeded` task also tests all the projects from the project lib dependencies of the testRuntime configuration.

Example 48.30. 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 48.31. 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. If the root project has declared `dep` (as this one does), then any task you run against the root project will cause that same named task to be run on all the children. So you can just run `gradle build` to build and test all projects.

48.9. Property and method inheritance

Properties and methods declared in a project are inherited to all its subprojects. This is an alternative to configuration injection. But we think that the model of inheritance does not reflect the problem space of multi-project builds very well. In a future edition of this user guide we might write more about this.

Method inheritance might be interesting to use as Gradle's *Configuration Injection* does not support methods yet (but will in a future release).

You might be wondering why we have implemented a feature we obviously don't like that much. One reason is that it is offered by other tools and we want to have the check mark in a feature comparison :). And we like to offer our users a choice.

48.10. Summary

Writing this chapter was pretty exhausting and reading it might have a similar effect. Our final message for this chapter is that multi-project builds with Gradle are usually *not* difficult. There are six elements you need to remember: `allprojects`, `subprojects`, `dependsOn`, `childrenDepe`, `dependsOnChildren` and project lib dependencies. [22] With those elements, and keeping in mind that Gradle has a distinct configuration and execution phase, you have already a lot of flexibility. But when you enter steep territory Gradle does not become an obstacle and usually accompanies and carries you to the top of the mountain.

---

[19] The real use case we had, was using http://lucene.apache.org/solr, where you need a separate war for each index your are accessing. That was one reason why we have created a distribution of webapps. The Resin servlet container allows us, to let such a distribution point to a base installation of the servlet container.

[20] `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.

[21] 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.

[22] So we are well in the range of the 7 plus 2 Rule :)

49

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 16, *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.

49.1. 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/(` 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 keeps separate the task declaration - that is, what the task should do - from the task implementation - that is, how the task does it.

See Chapter 51, *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.

49.2. Writing a simple task class

To implement a custom task class, you extend `DefaultTask`.

Example 49.1. 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 49.2. 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 `greet` property, and set the value when we declare the `greeting` task.

Example 49.3. 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 {
    def 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
```

49.3. 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 49.4. A build for a custom task

`build.gradle`

```
apply plugin: 'groovy'

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

Note: The code for this example can be found at `samples/customPlugin/plugin` which is in both the binary and source distributions of Gradle.

We just follow the convention for where the source for the task class should go.

Example 49.5. 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
    }
}
```

49.3.1. 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 51.5, "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 49.6. 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-SNAI
    }
}

task greeting(type: org.gradle.GreetingTask) {
    greeting = 'howdy!'
}
```

49.3.2. 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 49.7. 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)
    }
}
```

50

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.

50.1. 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/gro` 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 51, *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.

50.2. 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 50.1. A custom plugin

`build.gradle`

```
apply plugin: GreetingPlugin

class GreetingPlugin implements Plugin<Project> {
    def 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.

50.3. 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 50.2. 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.greeting = new 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 `mess` . 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 50.3. 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.greeting = new GreetingPluginExtension()
        project.task('hello') << {
            println "${project.greeting.message} from ${project.greeting.greete
        }
    }
}

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.

50.3.1. Using extensions for default values

The extension mechanism is also a powerful way of declaring default values for objects such as tasks. Furthermore, these default values can be specified in terms of other properties.

Example 50.4. A task with a configuration property

`build.gradle`

```
class GreetingTask extends DefaultTask {

    String greeting

    @TaskAction
    def greet() {
        println getGreeting()
    }
}
```

Given the above task, we can wire in a default value for the `greeting` property that is any value.

In this case we defer to a project property of the same name.

Example 50.5. Wiring in the task property default value with conventions

`build.gradle`

```
class GreetingPlugin implements Plugin<Project> {
    def void apply(Project project) {
        project.tasks.withType(GreetingTask) { task ->
            task.conventionMapping.greeting = { project.greeting }
        }
    }
}
```

By using the convention mapping above to map the value of the project property `greeting` as the value for the `greeting` property on all `GreetingTask` tasks, we have effectively configured this as the default value. That is, individual tasks can be overridden in such a way to override this default.

Example 50.6. Overriding conventional defaults

`build.gradle`

```
apply plugin: GreetingPlugin

// our default greeting
greeting = "Hello!"

task hello(type: GreetingTask)

task bonjour(type: GreetingTask) {
    greeting = "Bonjour!"
}
```

In the above, the `hello` task will assume the default value, while `bonjour` overrides this explicitly.

Example 50.7. Conventional defaults in action

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

```
> gradle -q hello bonjour
Hello!
Bonjour!
```

Note that the convention mapping is “live” in that the convention mapping closure will be evaluated everytime that the value is requested. In this example this means that the default value for the task property will always be the value of `project.greeting`, no matter when or how it changes.

50.4. 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 50.8. 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
}

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 above example 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.

50.5. 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 50.9. A build for a custom plugin

`build.gradle`

```
apply plugin: 'groovy'

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

Note: The code for this example can be found at `samples/customPlugin/plugin` which is in both the binary and source distributions 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 name of your plugin.

Example 50.10. Wiring for a custom plugin

`src/main/resources/META-INF/gradle-plugins/greeting.properties`

```
implementation-class=org.gradle.GreetingPlugin
```

Notice that the properties filename matches the plugin's name and is placed in the resources folder, and that the `implementation-class` property identifies the `Plugin` implementation class.

50.5.1. 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 51.5, "External dependencies for the build script". The following example shows how you might do this when the JAR containing the plugin has been published to a local repository:

Example 50.11. 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-SNAI
    }
}

apply plugin: 'greeting'
```

50.5.2. Writing tests for your plugin

You can use the `ProjectBuilder` class to create `Project` instances to use when you test your plugin implementation.

Example 50.12. Testing a custom plugin

`src/test/groovy/org/gradle/GreetingPluginTest.groovy`

```
class GreetingPluginTest {
    @Test
    public void greeterPluginAddsGreetingTaskToProject() {
        Project project = ProjectBuilder.builder().build()
        project.apply plugin: 'greeting'

        assertTrue(project.tasks.hello instanceof GreetingTask)
    }
}
```

50.6. Maintaining multiple domain objects

Gradle provides some utility classes for maintaining collections of object, which work well with the Gradle build language.

Example 50.13. 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> {
    def 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/custom
quickStart -> /home/user/gradle/samples/userguide/organizeBuildLogic/customPlug
userGuide -> /home/user/gradle/samples/userguide/organizeBuildLogic/customPlugi
```

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.container` methods, it MUST expose a property named “name” as the unique, and constant, name for the object. The `project.container(Class)` variant of the container method creates new instances by attempting to invoke the constructor of the class that takes a single string argument, which is the desired name of the object. See the above link for `proj` method variants that allow custom instantiation strategies.

51

Organizing Build Logic

Gradle offers a variety of ways to organize your build logic. First of all you can put your build logic directly in the action closure of a task. If a couple of tasks share the same logic you can extract this logic into a method. If multiple projects of a multi-project build share some logic you can define this method in the parent project. If the build logic gets too complex for being properly modeled by methods you want have an OO Model. [23] 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 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.

51.1. 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 51.1. Using inherited properties and methods

`build.gradle`

```
srcDirName = 'src/java'

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
```

51.2. Injected configuration

You can use the configuration injection technique discussed in Section 48.1, “Cross project configuration” and Section 48.2, “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 51.2. Using injected properties and methods

`build.gradle`

```
subprojects {
    // Inject a property and method
    srcDirName = 'src/java'
    srcDir = { file(srcDirName) }

    // Inject 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') {
    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
```

51.3. 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 51.1. Default buildSrc build script

```
apply plugin: 'groovy'
dependencies {
    compile gradleApi()
    groovy localGroovy()
}
```

This means that you can just put you build source code in this directory and stick to the layout convention for a Java/Groovy project (see Table 22.4, “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 51.3. Custom buildSrc build script

`buildSrc/build.gradle`

```
repositories {
    mavenCentral()
}

dependencies {
    testCompile group: 'junit', name: 'junit', version: '4.8.2'
}
```

The `buildSrc` project can be a multi-project build. This works like any other regular Gradle multi-project build. However, you need to make all of the projects that you wish be on the classpath of the actual build `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 51.4. 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` which is in both the binary and source distributions of Gradle.

51.4. 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 51.5. 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.
```

51.5. 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 51.6. 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 42.4, “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 51.7. 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.

51.6. 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. [24]

Example 51.8. Ant optional dependencies

build.gradle

```
configurations {
    ftpAntTask
}

dependencies {
    ftpAntTask("org.apache.ant:ant-commons-net:1.8.2") {
        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
            fileset(dir: "htdocs/manual")
        }
    }
}
```

This is also nice example for the usage of client modules. The pom.xml in maven central for the ant-commons-net task does not provide the right information for this use case.

51.7. 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.

---

[23] Which might range from a single class to something very complex.

[24] In fact, we think this is anyway the nicer solution. Only if your buildscript and Ant's optional task need the *same* library you would have to define it two times. In such a case it would be nice, if Ant's optional task would automatically pickup the classpath defined in the `gradesettings`.

52

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.

52.1. 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 customise 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 51.3, “Build sources in the `buildSrc` project” for details of this feature).

52.2. 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.

52.3. 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.

52.3.1. 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 52.1. 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]
```

52.4. External dependencies for the init script

In Section 51.5, “External dependencies for the build script” is was explained how to add external dependencies to a build script. Init scripts can similarly have external dependencies defined. You do this using the `initscript()` method, passing in a closure which declares the init script classpath.

Example 52.2. Declaring external dependencies for an init script

`init.gradle`

```
initscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath group: 'org.apache.commons', name: 'commons-math', version:
    }
}
```

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 42.4, “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 52.3. 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:
    }
}

println Fraction.ONE_FIFTH.multiply(2)
```

Output of **`gradle --init-script init.gradle -q doNothing`**

```
> gradle --init-script init.gradle -q doNothing
2 / 5
```

53

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 53.1. Wrapper task

`build.gradle`

```
task wrapper(type: Wrapper) {
    gradleVersion = '0.9'
}
```

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 53.2. 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 there is for example an improvement in the gradle-wrapper functionality you need to regenerate the wrapper files.

53.1. Configuration

If you run Gradle with **gradlew**, Gradle checks if a Gradle distribution for the wrapper is available. If not it tries to download it, otherwise it delegates to the **gradle** command of this distribution with all the arguments passed originally to the **gradlew** command.

You can specify where the wrapper files should be stored (within your project directory):

Example 53.3. Configuration of wrapper task

`build.gradle`

```
task wrapper(type: Wrapper) {
    gradleVersion = '0.9'
    jarFile = 'wrapper/wrapper.jar'
}
```

Build layout

```
customized/
  gradlew
  gradlew.bat
  wrapper/
    wrapper.jar
    wrapper.properties
```

You can specify the download URL of the wrapper distribution. You can also specify where the wrapper distribution should be stored and unpacked (either within the project or within the gradle user home dir). If the wrapper is run and there is local archive of the wrapper distribution Gradle tries to download it and stores it at the specified place. If there is no unpacked wrapper distribution Gradle unpacks the local archive of the wrapper distribution at the specified place. All the configuration options have defaults except the version of the wrapper distribution.

For the details on how to configure the wrapper, see `Wrapper`

If you don't want any download to happen when your project is build via **gradlew**, simply add the Gradle distribution zip to your version control at the location specified by your wrapper configuration. Relative url is supported - you can specify a distribution file relative to the location of g file.

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

53.2. Unix file permissions

The Wrapper task adds appropriate file permissions to allow the execution for the gradlew *NIX command. Subversion preserves this file permission. We are not sure how other version control systems deal with this. What should always work is to execute `sh gradlew`.

53.3. Environment variable

Some rather exotic use cases might occur when working with the Gradle Wrapper. For example the continuos integration server goes down during unzipping the Gradle distribution. As the distribution directory exists **gradlew** delegates to it but the distribution is corrupt. Or the zip-distribution was not properly downloaded. When you have no admin right on the continuous integration server to remove the corrupt files, Gradle offers a solution via environment variables.

Table 53.1. Gradle wrapper environment variables

Variable Name	Meaning
GRADLE_WRAPPER_ALWAYS_UNPACK	If set to `true`, the distribution directory gets always deleted when **gradlew** is run and the distribution zip is freshly unpacked. If the zip is not there, Gradle tries to download it.
GRADLE_WRAPPER_ALWAYS_DOWNLOAD	If set to `true`, the distribution directory and the distribution zip gets always deleted when **gradlew** is run and the distribution zip is freshly downloaded.

54

Embedding Gradle

54.1. 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. This API allows you to execute and monitor builds, and to query Gradle about the details of a build. The main audience for this API is IDE, CI server, other UI authors, 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 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 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.

The Tooling API is the official and recommended way to embed Gradle. This means that the existing APIs, namely `GradleLauncher` and the open API (the UIFactory and friends), are mildly deprecated and will be removed in some future version of Gradle. If you happen to use one of the above APIs, please consider changing your application to use the tooling API instead.

54.2. Quickstart

Since the Tooling API is an interface for a programmer most of the documentation lives in the javadocs/groovydocs. This is exactly our intention - we don't expect this chapter to grow very much. Instead we will increase code samples and improve the documentation in the javadocs. The main entry point to the Tooling API is the `GradleConnector`. You can navigate from there and find code samples and other instructions. Pretty effective way of learning how to use the Tooling API is checking out and running the *samples* that live in `$gradleHome/samples/toolingApi`.

If you're embedding gradle and you're looking for exact set of dependencies the tooling api jar requires please look at one of the samples at `$gradleHome/samples/toolingApi`. The dependencies are declared in the gradle build scripts. You can also find the repository declarations where the jars are obtained from.

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
`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/groovy-1.5.6`	Groovy project using Groovy 1.5.6
`groovy/groovy-1.6.7`	Groovy project using Groovy 1.6.7

`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.
`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`	Sala project using the Fast Scala Compiler (fsc).
`scala/mixedJavaAndScala`	A project containing a mix of Java and Scala source.
`scala/quickstart`	Scala quickstart project
`toolingApi/build`	An application which uses the tooling API to execute a Gradle build.
`toolingApi/eclipse`	An application which uses the tooling API to build the Eclipse model for a project.

`toolingApi/idea`	An application which uses the tooling API to extract information needed by IntelliJ IDEA.
`webApplication/customised`	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, `basicEc`). 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 'localScope1No
    try {localScope2} catch(MissingPropertyException e) {println 'localScope2No
    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. [25]

B.2. Configuration and execution phase

It is important to keep in mind that Gradle has a distinct configuration and execution phase (see Chapter 47, *The Build Lifecycle*).

Example B.2. Distinct configuration and execution phase

`build.gradle`

```
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.

---

[25] One of those discussions can be found here: http://groovy.329449.n5.nabble.com/script-scoping-question-td355887.html

C

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.

-C, --cache
: Specifies how compiled build scripts should be cached. Possible values are: `rebuild` or `on`. Default value is `on`. See Section 14.8, "Caching".

-D, --system-prop
: Sets a system property of the JVM, for example `-Dmyprop=myvalue`. See Section 14.2, "Gradle properties and system properties".

-I, --init-script
: Specifies an initialization script. See Chapter 52, *Initialization Scripts*.

-P, --project-prop
: Sets a project property of the root project, for example `-Pmyprop=myvalue`. See Section 14.2, "Gradle properties and system properties".

-S, --full-stacktrace
: Print out the full (very verbose) stacktrace for any exceptions. See Chapter 18, *Logging*.

-a, --no-rebuild
: Do not rebuild project dependencies.

--all
: Shows additional detail in the task listing. See Section 11.5.2, "Listing tasks".

-b, --build-file
: Specifies the build file. See Section 11.4, "Selecting which build to execute".

-c, --settings-file

Specifies the settings file.

`-d`, `--debug`
Log in debug mode (includes normal stacktrace). See Chapter 18, *Logging*.

`-e`, `--embedded`
Specify an embedded build script.

`-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`, `--info`
Set log level to info. See Chapter 18, *Logging*.

`-m`, `--dry-run`
Runs the build with all task actions disabled. See Section 11.6, “Dry Run”.

`--no-color`
Do not use color in the console output.

`-p`, `--project-dir`
Specifies the start directory for Gradle. Defaults to current directory. See Section 11.4, “Selecting which build to execute”.

`--profile`
Profiles build execution time and generates a report in the *`buildDir`*`/reports/profile` directory. See Section 11.5.5, “Profiling a build”.

`--project-cache-dir`
Specifies the project-specific cache directory. Default value is `.gradle` in the root project directory. See Section 14.8, “Caching”.

`-q`, `--quiet`
Log errors only. 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”.

`--continue`
Continues task execution after a task failure.

The above information is printed to the console when you execute **`gradle -h`**.

C.1. Deprecated command-line options

The following options are deprecated and will be removed in a future version of Gradle:

`-n`, `--dependencies`
(deprecated) Show list of all project dependencies. You should use **`gradle dependencies`** instead. See Section 11.5.3, “Listing project dependencies”.

`-r`, `--properties`
(deprecated) Show list of all available project properties. You should use **`gradle properti`** instead. See Section 11.5.4, “Listing project properties”.

`-t`, `--tasks`
(deprecated) Show list of available tasks. You should use **`gradle tasks`** instead. See Section 11.5.2, “Listing tasks”.

C.2. Daemon command-line options:

The Chapter 13, *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 13, *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`.

C.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 14.3, “Configuring gradle via gradle.properties” contains specific information about gradle configuration available via system properties.

C.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="-Dc` to use the Gradle daemon without needing to use the `--daemon` option every time you run Gradle. Section 14.3, “Configuring gradle 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.

D

Existing IDE Support and how to cope without it

D.1. IntelliJ

Gradle has been mainly developed with Idea IntelliJ and its very good Groovy plugin. Gradle's build script [26] 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.

D.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.

D.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.` 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 D.1. gradle-imports

```
import org.gradle.*
import org.gradle.util.*
import org.gradle.api.*
import org.gradle.api.artifacts.*
import org.gradle.api.artifacts.dsl.*
import org.gradle.api.artifacts.maven.*
import org.gradle.api.artifacts.specs.*
import org.gradle.api.execution.*
import org.gradle.api.file.*
import org.gradle.api.resources.*
import org.gradle.api.initialization.*
import org.gradle.api.invocation.*
import org.gradle.api.java.archives.*
import org.gradle.api.logging.*
import org.gradle.api.plugins.*
import org.gradle.plugins.ide.eclipse.*
import org.gradle.plugins.ide.idea.*
import org.gradle.plugins.jetty.*
import org.gradle.api.plugins.quality.*
import org.gradle.api.plugins.announce.*
import org.gradle.api.specs.*
import org.gradle.api.tasks.*
import org.gradle.api.tasks.bundling.*
import org.gradle.api.tasks.diagnostics.*
import org.gradle.api.tasks.compile.*
import org.gradle.api.tasks.javadoc.*
import org.gradle.api.tasks.testing.*
import org.gradle.api.tasks.util.*
import org.gradle.api.tasks.wrapper.*
import org.gradle.process.*
```

---

[26] 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
??