
Gradle

A build system

Version 0.4

Copyright ©2007-2008 Hans Dockter

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

Contents

1	Introduction	1
2	The Gradle Core Tutorial	2
2.1	Hello World	2
2.2	Build scripts are code	2
2.3	Task dependencies	3
2.4	Dynamic tasks	3
2.5	Manipulating existing tasks	3
2.6	Shortcut notations	4
2.7	Ant	4
2.8	Using methods	5
2.9	Default Tasks	5
2.10	Configure By DAG	6
2.11	Summary	6
3	The Java Projects Appetizer	7
3.1	Examples	7
4	The 'This And That' Tutorial	9
4.1	Skipping Tasks	9
4.2	Console Output	10
4.3	Directory Creation	10
4.4	Gradle Properties and System Properties	11
4.4.1	Checking for Project Properties	12
4.5	Accessing the web via a proxy	12
4.6	Caching	12
5	Overview	13
5.1	Features	13
5.2	Why Groovy?	14
5.3	Missing Features	14
6	The Project and Task API	15
6.1	Project API	15
6.2	Task API	15
6.3	Summary	15
7	More about Tasks	16
7.1	Configuring Tasks	16
7.2	Replacing Tasks	17
7.3	Summary	17
8	Plugins	18
8.1	Declaring Plugins	18
8.2	Configuration	19
8.2.1	More about convention objects	19
8.2.2	Declaring Plugins Multiple Times	20
8.3	Summary	20

9	The Java Plugin	21
9.1	Init	21
9.2	Javadoc	21
9.3	Clean	21
9.4	Resources	23
9.5	Compile	23
9.6	Test	23
9.7	Bundles	23
9.7.1	The libs Task	24
9.7.2	The dists Task	24
9.7.3	Adding Archives	24
9.8	Archive Tasks	25
9.8.1	Common Properties	25
9.8.2	Adding Content	26
9.8.3	Merging	26
9.8.4	Manifest	27
9.8.5	MetaInf	27
9.9	Upload	27
9.10	Eclipse	27
9.10.1	Eclipse Classpath	27
9.10.2	Eclipse Project	27
10	The Groovy Plugin	28
10.1	Compile	28
10.2	Test	29
11	The War Plugin	30
11.1	Default Settings	30
11.2	Customizing	30
11.3	Eclipse WTP	31
12	Dependency Management	32
12.1	Introduction	32
12.1.1	Versioning the jar name	32
12.1.2	Transitive dependency management	32
12.1.3	Version conflicts	33
12.1.4	Dependency management and Java	33
12.2	How to declare your dependencies	33
12.2.1	Configurations	33
12.2.2	Artifact Dependencies	33
12.2.3	Module Dependencies	34
12.2.4	Client Module Dependencies	34
12.2.5	Project dependencies	34
12.2.6	Global Excludes	34
12.2.7	Ivy dependencies	35
12.2.8	Configuring the Dependency Manager	35
12.3	Java Plugin and Dependency Management	35
12.4	Strategies of transitive dependency management	35
12.4.1	Enterprise Environments	36
12.4.2	Other environments	36
12.4.3	Implicit transitive dependencies	36
12.5	Repositories and Resolvers	37
12.5.1	Introduction	37
12.5.2	Maven Repository	37
12.5.3	Flat Directory Resolver	38
12.5.4	Cache	38
12.5.5	Resolver Container	39
12.5.6	More about Ivy resolvers	39

13 The Build Lifecycle	40
13.1 Build Phases	40
13.2 Settings File	40
13.3 Multi-Project Builds	41
13.3.1 Project Locations	41
13.3.2 Building the tree	41
13.3.3 Modifying Elements of the Project Tree	41
13.4 Initialization	42
13.5 Configuration and Execution of a Single Project Build	42
14 Multi-project builds	43
14.1 Cross Project Configuration	43
14.1.1 Defining Common Behavior	43
14.2 Subproject Configuration	44
14.2.1 Defining Common Behavior	44
14.2.2 Adding Specific Behavior	44
14.2.3 Project Filtering	45
14.3 Execution rules for multi-project builds	47
14.4 Running Tasks by there Absolute Path	48
14.5 Project and Task Paths	48
14.6 Dependencies - Which dependencies?	48
14.6.1 Execution Dependencies	49
14.6.2 Configuration Time Dependencies	51
14.6.3 Real Life examples	52
14.7 Project Lib Dependencies	53
14.8 Property and Method Inheritance	54
14.9 Summary	54
15 Organizing Build Logic	56
15.1 Build Sources	56
15.2 External dependencies	56
15.3 Ant Optional Dependencies	56
15.4 Summary	57
16 The Gradle Wrapper	58
16.1 Configuration	58
16.2 Unix file permissions	58
16.3 Environment variable	59
A Potential Traps	60
A.1 Groovy Script Variables	60
A.2 Configuration and Execution Phase	60
B Existing IDE Support and how to cope without it	62
B.1 IntelliJ	62
B.2 Eclipse	62
B.3 Using Gradle without IDE support	62
C Command line	63

Chapter 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!
- a very powerful support for multi-project builds.
- a very powerful dependency management (based on Apache Ivy).
- full support for your existing Maven or Ivy repository infrastructure.
- (optionally) support for transitive dependency management without the need for remote repositories or `pom.xml` and `ivy.xml` files.
- Ant tasks as first class citizens.
- *Groovy* build scripts.

In chapter 5 you find a detailed overview of Gradle. Now the tutorials are waiting, have fun :)

Chapter 2

The Gradle Core Tutorial

If you execute a build, Gradle normally looks for a file called `build.gradle` in the current directory (There are command line switches to change this behavior. See Appendix C). We call the `build.gradle` a build script. Although strictly speaking it is a build configuration script, as we see later. In Gradle the location of the build script file defines a project. The name of the directory containing the build script is the name of the project.

2.1 Hello World

In Gradle everything revolves around tasks. The tasks for your build are defined in the build script. To try this out, create the following build script named `build.gradle`.

```
createTask('hello') {
    println 'Hello world!'
}
```

Enter with your shell into the containing directory and execute the build script¹²:

```
current dir: userguide/tutorial/hello
>gradle -q hello
Hello world!
```

If you think this looks damn 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 to 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, like `javac` or `copy`, task. So if 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.

2.2 Build scripts are code

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

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

```
current dir: userguide/tutorial/upper
>gradle -q upper
Original: mY_nAmE
Upper case: MY_NAME
```

or

```
createTask('count') {
    4.times { print "$it " }
}
```

¹Every code sample in this chapter can be found in the `samples` dir of your Gradle distribution. The output box always denotes the directory name relative to the `samples` dir.

²The scripts are executed with the `-q` options which suppresses the Gradle logging.

```

current dir: userguide/tutorial/count
>gradle -q count
0 1 2 3

```

2.3 Task dependencies

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

```

createTask('hello') {
    println 'Hello world!'
}
createTask('intro', dependsOn: 'hello') {
    println "I'm Gradle"
}

```

```

current dir: userguide/tutorial/intro
>gradle -q intro
Hello world!
I'm Gradle

```

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

```

createTask('taskX', dependsOn: 'taskY') {
    println 'taskX'
}
createTask('taskY') {
    println 'taskY'
}

```

```

current dir: userguide/tutorial/lazyDependsOn
>gradle -q taskX
taskY
taskX

```

The dependency of targetX to targetY is declared before targetY is created. This is very important for multi-project builds.

2.4 Dynamic tasks

The power of Groovy can not only be used inside the tasks. You can use it for example to dynamically create tasks.

```

4.times { counter ->
    createTask("task_${counter}") {
        println "I'm task number $counter"
    }
}

```

```

current dir: userguide/tutorial/dynamic
>gradle -q task_1
I'm task number 1

```

2.5 Manipulating existing tasks

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

```

4.times { counter ->
    createTask("task_${counter}") {
        println "I'm task number $counter"
    }
}
task('task_0').dependsOn 'task_2', 'task_3'

```

```

current dir: userguide/tutorial/dynamicDepends
>gradle -q task_0
I'm task number 2
I'm task number 3
I'm task number 0

```

Or you can add behavior to an existing task.

```

createTask('hello') {
    println 'Hello Earth'
}
task('hello').doFirst {
    println 'Hello Venus'
}
task('hello').doLast {
    println 'Hello Mars'
}

```

```

current dir: userguide/tutorial/helloEnhanced
>gradle -q hello
Hello Venus
Hello Earth
Hello Mars

```

The calls `doFirst` and `doLast` can be executed multiple times. What they do is to add an action to the beginning or the end of the tasks actions list.

2.6 Shortcut notations

There is a convenient notation for accessing *existing* tasks.

```

createTask('hello') {
    println 'Hello world!'
}
hello.doLast {
    println 'Hello again!'
}

```

```

current dir: userguide/tutorial/helloWithShortCut
>gradle -q hello
Hello world!
Hello again!

```

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

2.7 Ant

Let's talk a little bit about Gradles Ant integration. Ant can be divided into two layers. The first layer is the Ant language. It contains the syntax for the `build.xml`, the handling of the targets, special constructs like `macrodefs`, etc. Basically everything except the Ant tasks and types. Gradle does not offer any special integration for this first layer. Of course you can in your build script execute an Ant build as an external process. Your build script may contain statements like: `"ant clean compile".execute()`³.

The second layer of Ant is its wealth of Ant tasks and types, like `javac`, `copy` or `jar`. For this layer Gradle provides excellent integration simply by relying on Groovy. Groovy is shipped with the fantastic `AntBuilder`. Using Ant tasks from Gradle is as convenient and more powerful than using Ant tasks from a `build.xml` file. Let's look at an example:

```

createTask('checksum') {
    File[] files = new File('../antChecksumFiles').listFiles()
    Arrays.sort(files)
}

```

³In Groovy you can execute Strings. To learn more about executing external processes with Groovy have a look in GINA 9.3.2 or at the Groovy wiki

```

files.each { File file ->
    ant.checksum(file: file.canonicalPath, property: file.name)
    println "$file.name Checksum: ${ant.antProject.properties[file.name]}"
}
}

```

```

current dir: userguide/tutorial/antChecksum
>gradle -q checksum
agile_manifesto.html Checksum: 2dd24e01676046d8dedc2009a1a8f563
agile_principles.html Checksum: 659d204c8c7ccb5d633de0b0d26cd104
dylan_thomas.txt Checksum: 91040ca1cefcbfdc8016b1b3e51f23d3

```

In your build script, a property called `ant` is provided by Gradle. It is a reference to an instance of Groovys `AntBuilder`. The `AntBuilder` is used the following way:

- Ant task names corresponds to `AntBuilder` method names.
- Ant tasks attributes are arguments for this methods. The arguments are passed in from of a map.
- Nested Ant tasks corresponds to method calls of the passed closure.

To learn more about the Ant Builder have a look in GINA 8.4 or at the Groovy Wiki

2.8 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.

```

createTask('checksum') {
    fileList('../antChecksumFiles').each { File file ->
        ant.checksum(file: file.canonicalPath, property: file.name)
        println "$file.name Checksum: ${ant.antProject.properties[file.name]}"
    }
}

createTask('length') {
    fileList('../antChecksumFiles').each { File file ->
        ant.length(file: file.canonicalPath, property: file.name)
        println "$file.name Length: ${ant.antProject.properties[file.name]}"
    }
}

File[] fileList(String dir) {
    File[] files = new File(dir).listFiles()
    Arrays.sort(files)
    files
}

```

```

current dir: userguide/tutorial/antChecksumWithMethod
>gradle -q checksum
agile_manifesto.html Checksum: 2dd24e01676046d8dedc2009a1a8f563
agile_principles.html Checksum: 659d204c8c7ccb5d633de0b0d26cd104
dylan_thomas.txt Checksum: 91040ca1cefcbfdc8016b1b3e51f23d3

```

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 15.

2.9 Default Tasks

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

```

defaultTasks "clean", "run"

createTask("clean") {
    println "Default Cleaning!"
}

createTask("run") {
    println "Default Running!"
}

createTask("other") {
    println "I'm not a default task!"
}

```

```

current dir: userguide/tutorial/defaultTasks
>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).

2.10 Configure By DAG

As we describe in full detail later (See chapter 13) 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 usecase 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.

```

version = null
createTask('init') {task, dag ->
    if (dag.hasTask(':release')) {
        version = '1.0'
    } else {
        version = '1.0-SNAPSHOT'
    }
}
createTask('distribution', dependsOn: 'init') {
    println "We build the zip with version=$version"
}
createTask('release', dependsOn: 'distribution') {
    println 'We release now'
}

```

```

current dir: userguide/tutorial/configByDag
>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).

2.11 Summary

This is not the end of the story for tasks. So far we have worked with simple tasks. Tasks will be revisited in chapter 7 and when we look at the Java Plugin in chapter 9.

Chapter 3

The Java Projects Appetizer

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

3.1 Examples

Provided your build script contains the single line

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

Executing `gradle libs` will compile, test and jar your code. If you specify a remote repository, executing `gradle uploadLibs` will do additionally an upload of your jar to a remote repository. Builds have usually more requirements. Let's look at a typical multi-project build.

Project Tree

```
D- ultimateApp
  D- api
  D- webservice
  D- shared
```

We have three projects. `api` is shipped to the client to provide them a Java client for your XML webservice. `webservice` is a webapp which returns XML. `shared` is code used both by `api` and `webservice`. Let's look at the Gradle build scripts of those projects.

ultimateApp

```
subprojects {
    manifest.mainAttributes([
        'Implementation-Title': 'Gradle',
        'Implementation-Version': '0.1'
    ])
    dependencies {
        compile "commons-lang:commons-lang:3.1"
        testCompile "junit:junit:4.4"
    }
    sourceCompatibility = 1.5
    targetCompatibility = 1.5
    test {
        include '**/*Test.class'
        exclude '**/Abstract*'
    }
}
```

The commons stuff for all Java projects we do define in the root project. Not by inheritance but via *Configuration Injection*. The root project is like a container and `subprojects` iterates over the elements of this container and injects the specified configuration. This way we can easily define the manifest content for all archives.

api

```
dependencies {
    compile "commons-httpclient:commons-httpclient:3.1", project(":shared")
}

dists {
```

```
zip() {  
    files(dependencies.resolve("runtime")) // add dependencies to zip  
    fileset(dir: "path/distributionFiles")  
}  
}
```

In the `api` build script we add further dependencies. One dependency are the jars of the `shared` project. Due to this dependency `shared` gets now always build before `api`. We also add a distribution, that gets shipped to the client.

Chapter 4

The 'This And That' Tutorial

4.1 Skipping Tasks

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

```
createTask('skipMe') {
    println 'This should not be printed if the mySkipProperty system property is set to true.'
}.skipProperties << 'mySkipProperty'
```

```
current dir: userguide/tutorial/skipProperties
>gradle -q -DmySkipProperty skipMe
```

The `-D` option of the `gradle` command has the same effect as the `-D` option of the `java` command. This way you can set system properties of the JVM that runs Gradle. You can add one or more *skip properties* to any existing task. If the corresponding system property is set to any value/footnoteThe statement `-Dprop` sets the property to empty string, thus you don't need to type more to skip a task. except `false` (case does not matter), the actions of the task don't get executed. But often you don't even need to set the skip properties. If you set a system property according to the pattern `skip.taskname`, the actions of this task don't get executed.

```
createTask('autoskip') {
    println 'This should not be printed if the skip.autoskip system property is set.'
}
```

```
current dir: userguide/tutorial/autoskip
>gradle -q -Dskip.autoskip autoskip
```

If you want tasks to be skipped, that depends on a skipped task, you have to declare this explicitly via the skip properties

```
createTask('autoskip') {
    println 'This should not be printed if the skip.autoskip system property is set.'
}
createTask('depends', dependsOn: 'autoskip') {
    println "This should not be printed if the skip.autoskip system property is set."
}.skipProperties << 'skip.autoskip'
```

```
current dir: userguide/tutorial/autoskipDepends
>gradle -q -Dskip.autoskip depends
```

If the rules for skipping a task can't be expressed with a simple property, you can use the `StopExecutionException`¹. 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.

```
createTask('compile') {
    println 'We are doing the compile.'
}

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

¹The fully qualified name is `org.gradle.api.tasks.StopExecutionException`

```

}
createTask('myTask', dependsOn: 'compile') {
    println 'I am not affected'
}

```

```

current dir: userguide/tutorial/stopExecutionException
>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.

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 ??).

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

```

createTask('disableMe') {
    println 'This should not be printed if the task is disabled.'
}.enabled = false

```

```

current dir: userguide/tutorial/disableTask
>gradle -q disableMe

```

4.2 Console Output

All available command line options are listed in Appendix C. You can also print them to your console with `gradle -h`. We want to look at the *console output* options in more detail.

Loglevel

Option	Meaning
neither <code>q</code> nor <code>d</code>	Gradle and Ant will print out info log messages.
<code>q</code>	Only errors are printed to the console.
<code>d</code>	Gradle and Ant will print out info and debug log messages.

Stacktraces

Option	Meaning
neither <code>s</code> nor <code>f</code>	No stacktraces are printed to the console in case of a build error (e.g. a compile error). Only in case of internal exceptions will stacktraces be printed. If the loglevel option <code>d</code> is chosen, truncated stacktraces are always printed.
<code>s</code>	Truncated stacktraces are printed. We recommend this over full stacktraces. Groovy full stacktraces are extremely verbose (Due to the underlying dynamic invocation mechanisms. Yet they usually do not contain relevant information for what has gone wrong in <i>your</i> code.)
<code>f</code>	The full stacktraces are printed out.

4.3 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):

```

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

```

```

current dir: userguide/tutorial/makeDirectory
>gradle -q compile
The class directory exists. I can operate

```

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

```

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

```

```

current dir: userguide/tutorial/directoryTask
>gradle -q otherResources
The class directory exists. I can operate

```

A *Directory Task* is a simple task which name is a relative path to the project dir². 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.

4.4 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.

There is also the possibility to directly add properties to your project objects. You can place a `gradle.properties` file either in the Gradle user home dir (defaults to `USER_HOME/.gradle`) or in your project dir. For multiproject builds you can place `gradle.properties` files in any subproject. The properties of the `gradle.properties` can be accessed via the project object. The properties file in the 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).³ If the environment property follows the pattern `ORG_GRADLE_PROJECT_yourProperty=somevalue`, `yourProperty` 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 prefix, which is `org.gradle.project..`

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).

```

gradle.properties
gradlePropertiesProp=gradlePropertiesValue
systemPropertiesProp=shouldBeOverWrittenBySystemProp
envPropertiesProp=shouldBeOverWrittenByEnvProp
systemProp.system=systemValue

```

```

properties - build.gradle
createTask('printProps') {
    println commandLineProjectProp
    println gradlePropertiesProp
    println systemProjectProp
    println envProjectProp
    println System.properties['system']
}

```

²The notation `dir('/somepath')` is a convenience method for `createTask('somepath', type: Directory)`

³*Teamcity* or *Bamboo* are for example CI servers which offer this functionality

```
current dir: properties
>gradle -q -PcommandLineProjectProp=commandLineProjectPropValue -Dorg.gradle.project.systemProjectProp=
commandLineProjectPropValue
gradlePropertiesValue
systemPropertyValue
envPropertyValue
systemValue
```

4.4.1 Checking for Project Properties

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

4.5 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 4.4. Your `gradle.properties` file could look like this:

```
systemProp.http.proxyHost=http://www.somehost.org
systemProp.http.proxyPort=8080
systemProp.http.proxyUser=userid
systemProp.http.proxyPassword=password
```

We could not find a good overview for all possible proxy settings. The best we can offer are the constants in a file from the ant project. Here a [link](#) to the svn view. If anyone knows a better overview please let us know via the mailing list.

4.6 Caching

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

Chapter 5

Overview

5.1 Features

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

Language for Dependency Based Programming This is the core of Gradle. Most build tools do offer such a thing. You can create tasks, create dependencies between them and those tasks get executed only once and in the right order. Yet compared to Ant¹ Gradle's task offer a rich API and can be any kind of object. Gradle's tasks support multi-project builds. There is much more to say about tasks later on.

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

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

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

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

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

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

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

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

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

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

¹We mean Ant's targets here.

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

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

5.2 Why Groovy?

We think the advantages of an internal DSL (based on a dynamic language) over XML are tremendous in case of *build scripts*. There are a couple of dynamic languages out there. Why Groovy? The answer lies in the context Gradle is operating in. Although Gradle is a general purpose build tool at its core, its main focus are Java projects.² In such projects obviously the team members know Java. One problem we see with Ant³ and Maven is, that it involves a lot of knowledge only available to the build master. Such builds are very hard to comprehend, let alone to modify by a person not deeply involved with those tools. We think a build should be as transparent as possible to *all* team members.

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

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

5.3 Missing Features

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

- Creating IDE project and classpath files. This is one of the top priorities for the next release. We want to provide this for IntelliJ, Eclipse and NetBeans.
- Integration for Emma, Cobertura and JDepends in our build-by-convention framework. Right now you have to integrate them yourself (for example with the Ant tasks for those tools).
- Integration of Jetty in our build-by-convention framework to allow easy startup of a web application.
- Integration of TestNG in our build-by-convention framework.

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

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

⁴At <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.

Chapter 6

The Project and Task API

6.1 Project API

In the tutorial in chapter 2 we have called for example the method `createTask`. Where does this method come from? We said earlier that the directory containing the build script defines a project for Gradle. For Gradle this means, that it creates an instance of `org.gradle.api.Project` and associates it with the build script. With the build script you can configure 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.

```
createTask('check') {  
    println project.name  
    println name  
}
```

```
current dir: userguide/tutorial/projectApi  
>gradle -q check  
projectApi  
projectApi
```

Both `println` statements print out the same property. One uses the auto delegation to the project object, for properties not defined in the build script. The other statement uses the `project` property available to any build script, which provides an instance of 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. Look here to learn more about [org.gradle.api.Project](#).

6.2 Task API

Many of the methods of the project object return task objects. We have already seen some ways on how to use the task objects in chapter 2. Look here to learn more about [org.gradle.api.Task](#).

6.3 Summary

The project and the task API constitute the core layer of Gradle and provide all the possible interaction options with this layer.¹ This core-layer constitutes a language for dependency based programming.² There are many other projects providing such a language. There is Ant for Java, Rake and Rant for Ruby, SCons for Python, the good old Make and many more.³ We think that one thing that makes Gradle special compared to the other tools, is its strong support for applying dependency based programming on `multi-project` builds. We also think that just Gradle's core layer (together with its integration of the Ant tasks), provides a more convenient build system than Ant's core layer.

¹There is more to come for this layer in the other chapters, e.g. support for multi-project builds (see chapter 14).

²Martin Fowler has written about this: <http://martinfowler.com/articles/rake.html#DependencyBasedProgramming>

³Interestingly, Maven2 is the only major build system which does not use dependency based programming.

Chapter 7

More about Tasks

In the introductory tutorial (chapter 2) you have learned how to create simple task. You have also learned how to add additional behavior to those 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. Tasks which have there own properties and methods. This is really different to what you are used from Ant targets. Such enhanced tasks are either provided by you or are provided by Gradle.

7.1 Configuring Tasks

As an example, let's look at the Resources task provided by Gradle. To create such a Resources task for your build, you may declare in your build script¹:

```
createTask('resources', type: org.gradle.api.tasks.Resources)
```

The resources task provides an API to configure it. See [Resources Task API](#). If you create the resources task like this, it has no default behavior.² We want now use the resources task to learn about different ways to configure it (the result is always the same for our examples).

```
Resources resources = createTask('resources', type: Resources)
resources.from(file('resources'))
resources.to(file('target'))
resources.includes('**/*.txt', '**/*.xml', '**/*.properties')
```

This is similar to the way we would normally configure objects in Java. You have to repeat the context (resources) 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.

```
Resources resources = createTask('resources', type: Resources)
resources.from(file('resources')).to(file('target')).
    includes('**/*.txt', '**/*.xml', '**/*.properties')
```

You might know this approach from the Hibernate's Criteria Query API or JMock. Of course the API of a task has to support this. The from, to and includes methods all return an instance of the resources object. All of Gradles build-in tasks usually support this configuration style.

But there is yet another way of configuring the resources task. It also preserves the context and it possibly offers the best readability. It is usually our favorite.

```
1 Resources resources = createTask('resources', type: Resources)
2 resources {
3     from(file('resources'))
4     to(file('target'))
5     includes('**/*.txt', '**/*.xml', '**/*.properties')
6 }
```

This works for *any* task. Line 2 of the example is just a shortcut for `task("resources")`. It is important to note that if you pass a closure to the `task` method, this closure is applied for *configuring* the task. There is a slightly different ways of doing this.

¹If you use the JavaPlugin, this task is automatically created and added to your project

²This is different when added by the JavaPlugin

```
Resources resources = createTask('resources', type: Resources).configure {
    from(file('source'))
    to(file('target'))
    includes('**/*.txt', '**/*.xml', '**/*.properties')
}
```

If you pass a closure to the `createTask()` method, this closure gets added as an *action* to the task. Every task has a `configure` method, which you can pass a closure for configuring the task. The above example works, because the `createTask` method returns the task object. Gradle uses this style for configuring objects in many places, not just for tasks.

7.2 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:

```
createTask('resources', type: Resources)

createTask('resources', overwrite: true) {
    println('I am the new one.')
}
```

```
current dir: userguide/tutorial/replaceTask
>gradle -q resources
I am the new one.
```

Here we replace a task of type `Resources` 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.

7.3 Summary

If you are coming from Ant, such an enhanced Gradle task as *Resources* 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.

Chapter 8

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. Gradle provides them via plugins. To make this clear at the very beginning. The plugins provided by Gradle belong to the core distribution of Gradle and they are NEVER updated or changed in any way for a particular Gradle distribution. If there is a bug in the compile functionality of Gradle, we gonna release a new version of Gradle. There is no change of behavior for the lifetime of a given distribution of Gradle. We mention this as there is another build tools with a plugin system which behaves differently.

8.1 Declaring Plugins

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

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

at the beginning of your script. That's all. From a technological point of view plugins use just the same operations as you can use from your build scripts. That is they use the Project and Task API (see chapter 6). The Gradle plugins use this API for:

- Adding tasks to your build (e.g. compile, test)
- Creating dependencies between those tasks to let them execute in the appropriate order.
- Adding a so called *convention object* to your project configuration.

Let's check this out:

```
usePlugin('java')

createTask('check') {
    println(task('compile').destinationDir.name) // We could also write println(compile)
}
```

```
current dir: userguide/tutorial/pluginIntro
>gradle -q check
classes
```

The JavaPlugin adds a `compile` task to the project object which can be accessed by a build script. The `usePlugin` method either takes a string or a class as an argument. You can write¹

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

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

```
java=org.gradle.api.plugins.JavaPlugin
groovy=org.gradle.api.plugins.GroovyPlugin
war=org.gradle.api.plugins.WarPlugin
```

If you want to use your own plugins, you must make sure that they are accessible via the build script classpath. See chapter 15 for more information.

¹Thanks to Gradles default imports (see B) you can also write `usePlugin(JavaPlugin)` in this case.

8.2 Configuration

If you use the *JavaPlugin* there is for example a compile and resources task for your production code (the same is true for your test code). The default location for the output of those task is the directory `build/classes`. What if you want to change this? Let's try:

```
usePlugin('java')

createTask('check') {
    resources.destinationDir = new File(buildDir, 'output')
    println(resources.destinationDir.name)
    println(compile.destinationDir.name)
}
```

```
current dir: userguide/tutorial/pluginConfig
>gradle -q check
output
classes
```

Setting the `destinationDir` of the resources task had only an effect on the resources 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.

Gradles tasks are usually *convention aware*. A plugin can add a convention object to your build. It can also map certain values of this convention object to task properties.

```
usePlugin('java')

createTask('check') {
    classesDirName = 'output'
    println(resources.destinationDir.name)
    println(compile.destinationDir.name)
    println(convention.classesDirName)
}
```

```
current dir: userguide/tutorial/pluginConvention
>gradle -q check
output
output
output
```

The *JavaPlugin* has added a convention object with a `classesDirName` property. The properties of a convention object can be accessed like project properties. As shown in the example, you can also access the convention object explicitly.

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.

8.2.1 More about convention objects

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.

```
usePlugin('java')
println classesDir
println convention.classesDir
println convention.plugins.java.classesDir
```

All three statements print out the same property. The more specific statements are useful if there are ambiguities.

8.2.2 Declaring Plugins Multiple Times

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

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

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

8.3 Summary

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

Chapter 9

The Java Plugin

Table 9.1 shows the default project layout assumed by the Java Plugin. This is configurable via the convention object. Table 9.2 shows the tasks added by the Java Plugin. These tasks constitute a lifecycle for Java builds. Table 9.3 and Table 9.5 shows the most important properties of the convention object of the Java plugin.¹ Gradle's conventions contain a convention for the directory hierarchy as well as conventions for the element names of the hierarchy. For example the `srcDirs` are relative to the `srcRoot`. Therefore `srcDirs` is a read-only property. If you want to change the name of the source dirs you need to do this via the `srcDirNames` property. But the paths you specify here are *relative* to the `srcRoot`. This has the advantage to make bulk changes easy. If you change `srcRoot` from `src` to `source`, this automatically applies to all directory properties which are relative to `srcRoot`. As this also introduces an inflexibility, we have additional floating dirs, which are not bound to any hierarchy (see Table 9.4). For example code generation tool could make use of this, by adding a source dir which is located in the build folder.

9.1 Init

The `init` task has no default action attached to it. It is meant to be a hook. You can add actions to it or associates your custom tasks with it. The Java Plugin executes this task before any other of its tasks get executed (except `clean` and `javadoc` which does not depends on `init`).

9.2 Javadoc

The `javadoc` task has no default association with any other task. It has no prerequisites on the actions of other tasks, as it operates on the source. It does not provide a fine-grained configuration API yet. If you need this, you have to use for example Ant's `javadoc` task directly.

Convention to Property Mapping	
Task Property	Convention Property
<code>srcDirs</code>	<code>srcDirs</code>
<code>destinationDir</code>	<code>[javadocDir]</code>

9.3 Clean

The `clean` task simply removes the directory denoted by its `dir` property. This property is mapped to the `buildDir` property of the project. In future releases there will be more control of what gets deleted. If you need more control now, you can use the *Ant delete task*.

¹The `buildDir` property is a property of the project object. It defaults to `build`.

Folder	Meaning
<code>src/main/java</code>	Application/Library sources
<code>src/main/resources</code>	Application/Library resources
<code>src/main/webapp</code>	Web application sources
<code>src/test/java</code>	Test sources
<code>src/test/resources</code>	Test resources

Table 9.1: Default Directory Layout

Taskname	dependsOn	Type
clean	-	org.gradle.api.tasks.Clean
javadoc	-	org.gradle.api.tasks.Javadoc
init	-	org.gradle.api.tasks.DefaultTask
resources	initialize	org.gradle.api.tasks.Resources
compile	resources	org.gradle.api.tasks.compile.Compile
testResources	compile	org.gradle.api.tasks.Resources
testCompile	testResources	org.gradle.api.tasks.compile.Compile
test	testCompile	org.gradle.api.tasks.testing.Test
libs	test	org.gradle.api.tasks.bundling.Bundle
uploadLibs	libs	org.gradle.api.tasks.Upload
dists	uploadLibs	org.gradle.api.tasks.bundling.Bundle
uploadDists	dists	org.gradle.api.tasks.Upload

Table 9.2: Java Plugin Tasks

Dir Name	Dir File	Default Value Name	Default Value File
srcRootName	srcRoot	src	<i>projectDir/src</i>
srcDirNames	srcDirs	main/java	[<i>srcRoot/main/java</i>]
resourceDirNames	resourceDirs	main/resources	[<i>srcRoot/main/resources</i>]
testSrcDirNames	testSrcDirs	test/java	[<i>srcRoot/test/java</i>]
testResourceDirNames	testResourceDirs	main/resources	[<i>srcRoot/main/resources</i>]
srcDocsDirName	srcDocsDir	docs	<i>srcRoot/docs</i>
classesDirName	classesDir	classes	<i>buildDir/classes</i>
testClassesDirName	testClassesDir	test-classes	<i>buildDir/test-classes</i>
testResultsDir	testResultsDirName	test-results	<i>buildDir/test-results</i>
distsDirName	distsDir	dists	<i>buildDir/dists</i>
docsDirName	docsDir	docs	<i>buildDir/docs</i>
javadocDirName	javadocDir	javadoc	<i>buildDir/javadoc</i>

Table 9.3: Java Convention Object - Directory Hierarchy Properties

Property	Type	Default Value
floatingSrcDirs	List	empty
floatingResourceDirs	List	empty
floatingTestResourceDirs	List	empty
floatingTestResourceDirs	List	empty

Table 9.4: Java Convention Object - Floating Dir Properties

Property	Type	Default Value
sourceCompatibility	String	null
targetCompatibility	String	null
manifest	GradleManifest	empty
metaInf	List	empty

Table 9.5: Java Convention Object - Non Dir Properties

9.4 Resources

The *Resources* task has two instances, `resources` and `testResources`.

Convention to Property Mapping		
Task Instance	Task Property	Convention Property
resources	sourceDirs	resourceDirs
resources	destinationDir	classesDir
testResources	sourceDirs	testResourceDirs
testResources	destinationDir	testClassesDir

The `resources` task offers includes and excludes directives as well as filters. Have a look at org.gradle.api.tasks.Resources to learn about the details.

9.5 Compile

The *Compile* task has two instances, `compile` and `testCompile`.

Convention to Property Mapping		
Task Instance	Task Property	Convention Property
compile	srcDirs	srcDirs
compile	destinationDir	classesDir
compile	sourceCompatibility	sourceCompatibility
compile	targetCompatibility	targetCompatibility
testCompile	srcDirs	testSrcDirs
testCompile	destinationDir	testClassesDir
testCompile	sourceCompatibility	sourceCompatibility
testCompile	targetCompatibility	targetCompatibility

The classpath of the compile task is derived from two sources. One is the *configuration* assigned to the task by the dependency manager. The other classpath source is the `unmanagedClasspath` property: a list of files denoting a jar or a directory. Usually you create your classpath with the dependency manager. The `unmanagedClasspath` is used internally by Gradle. This classpath is not shared between projects in a multi-project build. Nor is it part of a dependency descriptor if you choose to upload your library to a repository. See section ?? how the JavaPlugin glues the tasks with the dependency manager and see the whole chapter 12 how to use the dependency manager.

Have a look at org.gradle.api.tasks.compile.Compile to learn about the details. The compile task delegates to Ants javac task to do the compile. Via the compile task you can set most of the properties of Ants javac task.

9.6 Test

The `test` task executes the unit tests which have been compiled by the `testCompile` task.

Convention to Property Mapping	
Task Property	Convention Property
testClassesDir	testClassesDir
testResultsDir	testResultsDir
unmanagedClasspath	[classesDir]

Have a look at org.gradle.api.tasks.testing.Test to learn more. Right now the test results are always in XML-format. The task has a `stopAtFailuresOrErrors` property to control the behavior when tests are failing. Test *always* executes all tests. It stops the build afterwards if `stopAtFailuresOrErrors` is true and there are failing tests or tests that have thrown an uncaught exception.

Per default the tests are run in a forked JVM and the fork is done per test. You can modify this behavior by setting forking to false or set the forkmode to once.

The test task delegates to Ants junit task. TestNG is not supported yet. You can expect TestNG support in one of our next releases.

9.7 Bundles

The *Bundle* task has two instances, `libs` and `dists`. The Bundle task is a special animal. It is a container for archive tasks (jar, zip, ...).

Convention to Property Mapping		
Task Instance	Task Property	Convention Property
libs	tasksBaseName	project.archivesBaseName
dists	tasksBaseName	project.archivesBaseName

9.7.1 The libs Task

The `libs` task contains all the archive tasks which constitute the libraries needed to use your project as a library. Executing the `libs` task leads to the execution of all the contained archive tasks. In a multi-project build the archives which are produced by the `libs` task are available in the classpath of a dependent project. If you upload your project into a repository, those archives are part of the dependency descriptor. If you come from Maven you can have only one library jar per project. With Ivy you can have as many as you want. The Java plugin adds by default one jar archive to the `libs` task. The task name is by default `archive_jar` and can of course be manipulated by this name. This jar contains the content of the `classesDir`. This is the behavior you are used from Maven. If you are happy with that you usually don't have to touch this task. Except if you want to change the names of the generated archives.

The `libs` task depends on the `test` task and its archive tasks. The archive tasks assigned to the `libs` task depend by default on the `test` task as well. You can change this via the `childrenDependsOn` property of the `libs` task.

9.7.2 The dists Task

The `dists` task contains all the archive task that make up your distributions. For example a binary and a source distribution. The `dists` task has two purposes. One is providing a hook for distributions in the lifecycle. The other is for uploading distribution archives (the same is also true for the `libs` task.) The `dists` task depends on its archive tasks and the `libs` task. The contained archive tasks depend by default on the `libs` task as well. You can change this via the `childrenDependsOn` property of the `dists` task.

9.7.3 Adding Archives

The Bundle task has a couple of convenience methods for adding new archive tasks to it. Right now there are the methods (`jar`, `war`, `zip`, `tar`, `tarGz`, `tarBzip2`). They all behave in the same way.

```
usePlugin('java')
archivesBaseName = 'myArchiveName' // defaults to project.name
version = 1.0
dists {
    // Creates a task archive_zip which generates an archive 'myArchiveName-1.0.zip'
    zip() {
        fileSet(dir: 'somedir')
    }
}
println archive_zip.archiveName
```

current dir: userguide/tutorial/zip

```
>gradle -q init
myArchiveName-1.0.zip
```

This adds an archive task with the name `archive_zip`. It is important to distinguish between the name of the archive task and the name of the archive generated by the archive task. The name of the generated archive file is by default the name of the project. The default for naming generated archives can be changed with the `archivesBaseName` project property. The name of the archive can be changed at any time later on. You can pass a closure to the `zip()` method, which statements are applied against the newly created archive task object. In section 9.8 you will learn more about which statements you can apply against a particular archive task. You could for example change the name of the archive directly in the closure:

```
usePlugin('java')
version = 1.0
dists {
    // Creates a task archive_zip which generates an archive 'customName-1.0.zip'
    zip() {
        fileSet(dir: 'somedir')
        baseName = 'customName'
    }
}
```

Type	Meaning
<code>org.gradle.api.tasks.util.FileSet</code>	A set of files defined by a common baseDir and include/exclude patterns.
<code>org.gradle.api.tasks.util.ZipFileSet</code>	Extends FileSet with additional properties known from Ants zipfileset task.
<code>org.gradle.api.tasks.util.TarFileSet</code>	Extends ZipFileSet with additional properties known from Ants tarfileset task.
<code>org.gradle.api.tasks.util.FileCollection</code>	An arbitrary collection of files to the archive. In contrast to a FileSet they don't need to have a common basedir.
<code>org.gradle.api.tasks.util.AntDirective</code>	An arbitrary Ant resource declaration.

Table 9.6: Filecontainer for Archives

```
}
println archive_zip.archiveName
```

```
current dir: userguide/tutorial/zipWithCustomName
>gradle -q init
customName-1.0.zip
```

You can further customize the names by passing arguments to archive method of the bundle task:

```
usePlugin('java')
archivesBaseName = 'gradle'
version = '1.0'
dists {
    // Creates an archive task archive_wrapper_src_zip with
    // the archive name gradle-wrapper-1.0-src.zip
    zip(appendix: 'wrapper', classifier: 'src') {
        fileSet(dir: 'somedir')
    }
}
println archive_wrapper_src_zip.archiveName
```

```
current dir: userguide/tutorial/zipWithArguments
>gradle -q init
gradle-wrapper-1.0-src.zip
```

9.8 Archive Tasks

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

Type	Accepted file container	Extends
<code>org.gradle.api.tasks.bundling.Zip</code>	fileSet, fileCollection, zipFileSet	AbstractArchiveTask
<code>org.gradle.api.tasks.bundling.Tar</code>	fileSet, fileCollection, zipFileSet, tarFileSet	Zip
<code>org.gradle.api.tasks.bundling.Jar</code>	fileSet, fileCollection, zipFileSet	Zip
<code>org.gradle.api.tasks.bundling.War</code>	fileSet, fileCollection, zipFileSet	Jar

The following file containers are available: To learn about all the details have a look at the javadoc of the archive task class or the file container class itself.

9.8.1 Common Properties

The name of the generated archive is assembled from the task properties `baseName`, `classifier` and `extension` to: `baseName-project.version-classifier.extension`². The assembled name is accessible via the `archiveName` property. The `name` property denotes the name of the task, not the generated archive. An archive task has also a `customName` property. If this property is set, the `archiveName` property returns its value instead of assembling a name out of the properties mentioned above.

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

²If the classifier is empty the trailing - is not added to the name. The same is true for the `project.version` property

9.8.2 Adding Content

To add content to an archive you must add file container to an archive (see Table 9.6). You can add as many file container as you like. They behave pretty much the same as the Ant resources with similar names.

```
myZipTask.fileSet(dir: 'contentDir') {
    include('**/*.txt')
    exclude('**/*.gif')
}
```

You can add arbitrary files to an archive:

```
myZipTask.files('path_to_file1', 'path_to_file2')
```

Other examples:

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

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

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

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

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

9.8.3 Merging

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

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

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

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

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

There is another method for merging called `mergeGroup`.

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

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

9.8.4 Manifest

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

```
manifest.mainAttributes(Implementation-Title: "Gradle", Implementation-Version: $version)
```

You can also define sections of a manifest file.

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

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

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

9.8.5 MetaInf

The convention object of the JavaPlugin has a `metaInf` property pointing to a list of `FileSet` objects. With this 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)
```

9.9 Upload

The `Upload` task has two instances, `uploadLibs` and `uploadDists`. An easy way of describing there behavior, is that all archives added to the `libs` and `dists` bundle are uploaded by the corresponding upload task. An upload task uploads to the repositories assigned to it. If needed you have more control on what files get uploaded. Have a look at `org.gradle.api.tasks.Upload` to learn more.

9.10 Eclipse

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

9.10.1 Eclipse Classpath

`org.gradle.api.tasks.ide.eclipse.EclipseClasspath` has a default instance with the name `eclipseCp`. It generates a `.classpath` file.

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

9.10.2 Eclipse Project

`org.gradle.api.tasks.ide.eclipse.EclipseProject` has a default instance with the name `eclipseProject`. It generates a `.project` file.

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

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

Chapter 10

The Groovy Plugin

The Groovy Plugin extends the JavaPlugin. It can deal with pure Java projects¹, with mixed Java and Groovy projects and with pure Groovy projects. The Groovy plugin does not add any tasks. It modifies some of the tasks of the JavaPlugin and adds to the *Convention* object. See Table 10.1, Table 10.2 and Table 10.3. It also add a new dependency configuration `groovy`.

Gradle is written in Groovy and offers 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 are some examples how this works (the notation depends on your resolvers):

```
dependencies {
    addMavenRepo()
    groovy "org.codehaus.groovy:groovy-all:1.6-beta-1"
    // further declarations
}
```

```
dependencies {
    addFlatDirResolver('lib', new File(rootDir, 'lib'))
    clientModule(['groovy'], ":groovy-all:1.5.5") {
        dependency(":commons-cli:1.0")
        clientModule(":ant:1.7.0") {
            dependencies(":ant-junit:1.7.0:jar", ":ant-launcher:1.7.0")
        }
    }
}
```

All the Groovy source directories can contain Groovy *and* Java code. The Java source directories may only contain Java source code (and can of course be empty)²

10.1 Compile

The *GroovyCompile* task has two instances, `compile` and `testCompile`. The task type extends the `Compile` task (see section 10.1)

¹We 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 pure `javac`. In particular as you would have to supply a groovy jar for doing this.

²We are using the same conventions as introduced by Russel Winders Gant tool (<http://gant.codehaus.org>).

Folder	Meaning
<code>src/main/groovy</code>	Application/Library sources in Groovy
<code>src/test/groovy</code>	Test sources in Groovy

Table 10.1: Default Directory Layout (additional to the Java layout)

Dir Name	Dir File	Default Value Name	Default Value File
groovySrcDirNames	groovySrcDirs	main/groovy	[srcRoot/main/groovy]
groovyTestSrcDirNames	groovyTestSrcDirs	test/groovy	[srcRoot/test/groovy]

Table 10.2: Groovy Convention Object (extends JavaConvention).

Property	Type	Default Value
floatingGroovySrcDirs	List	empty
floatingGroovyResourceDirs	List	empty
floatingGroovyTestResourceDirs	List	empty
floatingGroovyTestResourceDirs	List	empty

Table 10.3: Groovy Convention Object (extends JavaConvention) - Floating Dir Properties

Additional Convention to Property Mapping		
Task Instance	Task Property	Convention Property
compile	groovySourceDirs	groovySrcDirs
testCompile	groovySourceDirs	groovyTestSrcDirs

Have a look at org.gradle.api.tasks.compile.GroovyCompile to learn about the details. 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.

/

10.2 Test

In contrast to the Java plugin the fork mode is set to once by default, because of the significant startup time of Groovy. The Java plugin uses per test as fork mode (see section 9.6).

Chapter 11

The War Plugin

The war plugin extends the JavaPlugin. It disables the default jar archive generation of the Java Plugin and adds a default war archive task. Have also a look at org.gradle.api.tasks.bundling.War.

11.1 Default Settings

The default behavior of the War plugin is to copy the content of `src/main/webapp` to the root of the archive. Your `webapp` folder may of course contain a `WEB-INF` sub-directory, which again may contain a `web.xml` file. Your compiled classes are compiled to `WEB-INF/classes`. All the dependencies of the `runtime`¹ configuration are copied to `WEB-INF/lib`. The War plugin add two new dependency configurations: `providedCompile` and `providedRuntime`. Those new configurations have the same scope as the respective `compile` and `runtime` configurations. Except that they are not added to the war-archive. It is important to note that those `provided` configurations work transitively. Let's say you add `commons-httpclient:commons-httpclient:3.0` to any of the provided configurations. This dependency has a dependency on `commons-codec`. This means neither `httpclient` nor `commons-codec` is added to your war, even if `commons-code` 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-httpclient:3.0@jar`.

11.2 Customizing

Here an example with the most important customization options:

```
group = 'gradle'
version = '1.0'
usePlugin('war')
targetCompatibility = '1.5'
sourceCompatibility = '1.5'

dependencies {
    addConfiguration('moreLibs')
    addFlatDirResolver('lib', "$rootDir/lib")
    compile ":compile:1.0"
    providedCompile ":providedCompile:1.0@jar"
    runtime ":runtime:1.0"
    providedRuntime ":providedRuntime:1.0@jar"
    testCompile ":junit:3.8.2"
    moreLibs ":otherConf:1.0"
}

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

¹The `runtime` configuration extends the `compile` configuration.

```
}
```

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

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

```
archive_jar.enabled = true
```

11.3 Eclipse WTP

[org.gradle.api.tasks.ide.eclipse.EclipseWtp](#) has a default instance with the name `eclipseWtp`. It generates a `.settings/org.eclipse.wst.common.component` file.

Chapter 12

Dependency Management

12.1 Introduction

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. But Gradle offers also a simpler approach which might be better suited for many projects.

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 there version as part of there file name.
- It has 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.

12.1.1 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 making it easier to maintain.

12.1.2 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 are loosing very soon 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 this jars belong to. 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 your first level dependency might also be a first level dependency itself. Or it might be a transitive dependency of another of your first level dependency. 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, nobody dares to remove any jar any longer. The project classpath is a complete mess and if a classpath problems arises, hell on earth invites you for a ride. In one of my former projects, I've found some ldap related jar in the classpath, which sheer presence, as I've 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.

12.1.3 Version conflicts

In your dependency description you tell Gradle which version of a dependency is needed by another dependency. This leads frequently 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 12.1.4). What Gradle offers you is a resolution strategy, by default the newest version is used. To deal with problems due to version conflicts, reports with dependency graphs are also very helpful. Such reports are another feature of dependency management.

12.1.4 Dependency management and Java

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

Both approaches rely on descriptor xml files, which contains information about the dependencies of a particular jar. Both also use repositories where the actual jars are placed together with there 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 *gigantic* 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 a very good 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-277. OSGi is available already, JSR-277 is supposed to be shipped with Java 7. These technologies deal 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.

12.2 How to declare your dependencies

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

12.2.1 Configurations

Dependencies are grouped in configurations. Configurations have a name and they can extend each other. If you use the Java or Groovy plugin, Gradle adds a number of related configurations to your build. The plugin also associates configurations with tasks. See section 12.3 for details. Of course you can add your own configurations on top of that. This is very handy for example for adding dependencies not needed for building or running your software (e.g. additional JDBC drivers). You might want to add such dependencies to a distribution of your software.

A dependency can belong to more than one configuration although that might be rather the exception. What is a dependency after all and how do you add it to a configuration?

12.2.2 Artifact Dependencies

To add an artifact dependency to let's say the `compile` configuration you simply type:

```
dependencies {
    compile "org.apache.ant:ant-junit:1.7.0:jar"
}
```

An artifact dependency is just a file, usually a jar file. The notation for such a dependency follows the pattern: `[group]:[artifact]:[version]@[extension]` or `[group]:[artifact]:[version]:[classifier]@[extension]`.¹ Gradle needs a repository somewhere where this artifact can be found, but there is no xml descriptor required in the repository. If there is one, it is ignored. In the simplest case such a repository is a folder on your local machine.

¹Some dependencies in the Maven repository use classifiers, for example TestNG. A declaration would look like this: `"org.testng:testng:5.8:jdk15@jar"`. But they are not very common.

12.2.3 Module Dependencies

Here is an example for a module dependency:

```
dependencies {
    compile "org.hibernate:hibernate:3.0.5"
}
```

The notation here is the same as for *artifact dependencies* except that you don't specify a file extension. If you declare a module dependency, Gradle looks in the repositories if there is either a Maven pom file or an ivy.xml file for this module. If there is no such XML file, Gradle either tries to download a jar with the same signature as the module dependency or throws an exception (or both, if this jar can't be found). It is configurable whether a XML descriptor for a module dependency is required. This is a per-repository configuration.

If the module descriptor (e.g. pom.xml) declares dependencies you don't want you have, you can exclude them. Gradle offers the following notation:

```
dependencies {
    addDependency(['compile'], "org.codehaus.groovy:groovy-all:1.5.4") {
        exclude(module: 'jline')
        exclude(module: 'junit')
    }
}
```

The `exclude` method simply delegates to `excludeRules.add()`. In subsection ?? you will find further details on exclude handling.

12.2.4 Client Module Dependencies

Client module dependencies enable you to declare *transitive* dependencies directly in your build script. Client module dependencies can be nested.

```
addClientModule(confs: ['compile'], id: "org.codehaus.groovy:groovy-all:1.5.4") {
    addDependency("commons-cli:commons-cli:1.0:jar")
    addClientModule("org.apache.ant:ant:1.7.0") {
        addDependency("org.apache.ant:ant-launcher:1.7.0:jar")
        addDependency("org.apache.ant:ant-junit:1.7.0")
    }
}
```

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.

In the current release *Client Modules* have one limitation. For example if 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 try to fix this for the next release of Gradle.

12.2.5 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*.

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

Multi-project builds are discussed in chapter 14.

12.2.6 Global Excludes

You can exclude a transitive dependency for all your dependencies via global excludes.

```
dependencies {
    excludeRules.add(org: 'junit', module: 'junit')
}
```

Name	Extends	Task	Meaning
compile	-	compile	Compile time dependencies
runtime	compile	-	Runtime dependencies
testCompile	compile	testCompile	Additional dependencies for compiling tests.
testRuntime	runtime, testCompile	test	Additional dependencies for running tests only.
libs	-	uploadLibs	Dependencies (e.g. jars) produced by this project.
default	runtime, master	-	Dependencies produced and required by this project.
dists	-	uploadDists	Archives added to the dists bundle belong to this configuration.

Table 12.1: Java/Groovy Configurations

You can either specify only the organization (group) or only the module name. Gradle delegates to Ivy's exclude handling for this. Ivy has a very powerful exclude handling including regex patterns. To make full use of this you can create an Ivy exclude rule by your self and add it to the exclude rules.

```
dependencies {
    excludeRules.getRules().add(new DefaultExcludeRule(new ArtifactId(
        new ModuleId('someorg', 'somemodule'),
        PatternMatcher.ANY_EXPRESSION,
        PatternMatcher.ANY_EXPRESSION,
        PatternMatcher.ANY_EXPRESSION),
        ExactPatternMatcher.INSTANCE, null));
}
```

See the Ivy documentation for more details.

12.2.7 Ivy dependencies

Gradle offers its own notation to describe dependencies. Under the hood we transform them into ivy dependency objects. Our notation provides only a subset of what is possible with Ivy. If you want or need to use options of Ivy not offered by our notation you can pass ivy dependency objects directly to Gradle.

```
DefaultDependencyDescriptor dependencyDescriptor = new DefaultDependencyDescriptor(
    new ModuleRevisionId(new ModuleId('junit', 'junit'), '4.2'), false)
// do more configuration with the descriptor
dependencies {
    addDependencyDescriptors dependencyDescriptor
}
```

12.2.8 Configuring the Dependency Manager

In the examples above we have already configured the dependency manager. How does this work? The `project` object offers a property called `dependencies` which points to an instance of org.gradle.api.DependencyManager. If you call `dependencies` with a closure, all statements within the closure are delegated first to the dependency manager. This is the same mechanism which is used for configuring tasks.

12.3 Java Plugin and Dependency Management

The Java Plugin preconfigures the dependency management. It adds a set of configurations and links the tasks like `compile` and `test` to this configuration. Table 12.1 shows the details. The Java plugin defines also where to find the jars and zips to be uploaded. Finally it does a couple of settings for multi-project builds. If you have for example declared a project dependency for the `compile` configuration you don't want this project to be build if you execute a `clean`.

12.4 Strategies of transitive dependency management

Many projects rely on the Maven2 repository. This is not without problems.

- The IBiblio repository is often down or has a very long response time.
- The pom.xml's of many projects have wrong informations. (as one example, the pom of commons-httpclient-3.0 declares junit as a runtime dependency).

- For many projects there is not one right set of dependencies (as more or less imposed by the pom format).

If your project relies on the IBiblio repository you are likely to need an additional custom repository, because:

- You might need dependencies that are not uploaded to IBiblio yet.
- You want to deal properly with wrong metadata in a IBiblio pom.xml.
- You don't want to expose people who want to build your project, to the relatively frequent downtimes or very long response times of IBiblio.

It is not a big deal to set-up a custom repository.² But it is 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. It is another layer of indirection, which contains information relevant for your build and you have to navigate to the lengthy path with your browser to get the information. All this is not really a big deal but in its sum it has a relevant impact on your agility. Why should you cope with this? You pay something, so you should also get something, right? So we end up with the question: Does it make sense to store the dependency information (and the dependencies) in remote repositories? Our answer is: It depends.

12.4.1 Enterprise Environments

In a larger enterprise environment you are likely to have a number of independent builds, possibly many different, large multi-project builds. Here a repository infrastructure makes a lot of sense. You have already infrastructure that guarantees a high uptime. Multiple projects can share their information about certain common dependencies they have. There is a central place to look for dependency information. And the repositories are used to integrate the internal projects with each other. One project publishes itself to the company repository, with all its dependency information stored in an ivy.xml or pom.xml file.³ Together with a continuous integration server this offers a strong integration for projects with separate builds. As you have seen, Gradle fully supports this approach.

12.4.2 Other environments

But what about other environments. What about a typical open source projects like Gradle or environments which consists of a single multi-project build. They don't integrate with other projects in the same environment or need to share dependency information in this environment. In such a (very common) case we think dependency management based on remote repositories is an unnecessary indirection which makes things more complicated than necessary. Gradle itself (which is build by Gradle) stores its dependencies in svn. There is a lib folder which contains all jars needed to build Gradle. Yet we have a complete management of our dependencies. For Gradle the lib folder is a local repository containing all the jars (in a flat structure). There are no xml descriptors. We use either client module dependencies to express our dependency relations, or artifact dependencies in case a first level dependency has no transitive dependencies. People can check out Gradle from svn and have everything necessary to build it. The same applies to our source distribution.

You might argue that such an approach works only for projects like Gradle, which is not used as a library but is simply a distribution. What about open source library projects (e.g. commons-httpclient) that need to upload themselves to the Maven2 repo together with a pom.xml file. As soon as the limitation mentioned in subsection 12.2.4 has been overcome, your project can upload to such a repository without relying on XML descriptor files themselves.

You could also have a mixed strategy. If your main concern is bad metadata in the 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.

12.4.3 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.

```
List groovy = ["org.codehaus.groovy:groovy-all:1.5.4:jar",
              "commons-cli:commons-cli:1.0:jar",
              "org.apache.ant:ant:1.7.0:jar"]
List hibernate = [...] // enter the dependencies here
```

²If you want to shield your project from the downtimes of IBiblio things get more complicated. You probably want to set-up a repository proxy for this. In an enterprise environment this is rather common. For an open source project it looks like overkill.

³As discussed in 12.4.2 this will be also possible with client modules in a future release.

```
dependencies {
    compile groovy
    runtime hibernate
}
```

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.

But 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.

12.5 Repositories and Resolvers

12.5.1 Introduction

Gradle (Ivy) distinguishes between *resolvers* and *repositories*. A repository is the physical source for your dependencies. A resolver is responsible for retrieving a dependency (usually from a repository but not necessarily). As a Gradle or Ivy user, *resolvers* are the objects you are working with. You can also aggregate multiple resolvers in a certain order (not just linked lists). It is one of the very strengths of Ivy, how versatile you can aggregate resolvers and that you can write easily your *own*. Gradle for example makes use of this to implement client modules. If you add a resolver, you always have to give it a name. This name is used to internally address a resolver but is also used for the dependency logging. The Java plugin does not add any default resolver to your projects dependency manager. Gradle offers a couple of convenience methods to add resolvers. You can also always add an instance of an Ivy resolver object directly.

12.5.2 Maven Repository

To add a resolver for the Maven2 repository (<http://repo1.maven.org/maven2>) simply type:

```
dependencies {
    addMavenRepo()
}
```

The added resolver has the name `MavenRepo` and is added to the `classpathResolvers` (see [12.5.5](#)).

If you want to add a custom repository with a Maven layout you can type:

```
dependencies {
    addMavenStyleRepo('myrepo', 'http://repo.gradle.org')
}
```

Quite often certain jars are not in the official Maven repository for licensing reasons (e.g. JTA). But the poms are there and you want to use them. Gradle (and Ivy) use the same resolver to look for the pom and the corresponding artifact. It is not possible to define a chain of resolvers, where one resolver is used for retrieving the pom and another resolver is used for retrieving the artifact. For the dependencies of the artifact the chain of resolvers can be used. To solve this problem

```
dependencies {
    addMavenRepo('http://repo.gradle.org', 'http://repo2.gradle.org')
}
```

Now Gradle looks only in the official Maven repository for the poms, but looks in both the official Maven repository and the two repositories specified above for the artifacts belonging to the pom. Let's look at an example to understand this better.

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>reptest</groupId>
  <artifactId>reptest</artifactId>
  <version>1.0</version>
  <dependencies>
    <dependency>
      <artifactId>testdep</artifactId>
      <groupId>testdep</groupId>
      <version>1.0</version>
```

```

    </dependency>
  </dependencies>
</project>

```

We have 4 files in different repositories. For this example let's assume that `repotest-1.0.pom` is in the official Maven repository and that the other files are in `http://repo.gradle.org`.

Let's define our resolvers like this:

```

dependencies {
  addMavenRepo()
  addMavenStyleRepo('myrepo', 'http://repo.gradle.org')
  compile "repotest:repotest:1.0"
}

```

Gradle finds `repotest-1.0.pom` in the official Maven repo. After this it tries to find the corresponding artifact `repotest-1.0.jar`. It looks only in the Maven repo for this file as it expects to find pom and artifact by the same resolver. The build fails.

Let's try to improve this:

```

dependencies {
  addMavenRepo('http://repo.gradle.org')
  compile "repotest:repotest:1.0"
}

```

Gradle finds `repotest-1.0.pom` in the Maven repo. After this it tries to find the corresponding artifact `repotest-1.0.jar` and finds it in `http://repo.gradle.org`. But what about `testdep-1.0.pom`? As Gradle looks for poms only in the Maven repo it can't find this pom file.⁴

The final correct version to solve our use case is:

```

dependencies {
  addMavenRepo('http://repo.gradle.org')
  addMavenStyleRepo('myrepo', 'http://repo.gradle.org')
  compile "repotest:repotest:1.0"
}

```

12.5.3 Flat Directory Resolver

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

```

dependencies {
  addFlatDirResolver('lib', "$rootDir/lib1", new File(rootDir, 'lib2'))
  dependencies(':junit:4.4', ':commons-io:1.3.1:jar')
}

```

In the example we assign multiple directories to a flat directory resolver⁵. The first argument is the name of the resolver. The group attributes in the dependency declaration is empty (but there has to be a colon at the beginning). Specifying the extension (e.g. `jar`) is optional and defaults to `jar`. You could add a group attribute if you are in a chain of resolvers. This resolver simply ignores it. The repositories could look like this:

```

project-root
- lib1
- junit-4.4.jar
- lib2
- commons-io-1.3.1.jar

```

The `addFlatDirResolver` methods adds a resolver to the `classpathResolvers` (see [12.5.5](#))

You can add easily file system repositories with a different layout (see [12.5.6](#))

12.5.4 Cache

When Gradle downloads dependencies from remote repositories it stores them in a local cache located at `USER_HOME/.gradle/cache`. When Gradle downloads dependencies from one of its predefined local resolvers (e.g. Flat Directory resolver), the cache is not used as an intermediate storage for dependencies.

⁴To make things more flexible (or complicated): By default the build would not fail, as Ivy's default behavior is to create internally a default pom if it can't find a pom. This default pom has of course no dependencies and defines just the artifact `testdep-1.0.jar` and would be found in `http://repo.gradle.org`. If you want the build to fail if a pom can't be found you must the set the `allowNoMdm` property of the resolver to `false`.

⁵You can pass any type for describing a path, as long as the type's `toString()` method returns a valid path.

12.5.5 Resolver Container

There are three hooks for adding resolvers. Via the resolvers assigned to the `classpathResolvers` property of the dependency manager, the `compile`, `testCompile` and `test` tasks retrieve their dependencies. The `uploadLibs` and `uploadDists` tasks have both an `uploadResolvers` property. You can assign one resolver to multiple hooks.

All of those hooks are instances of `org.gradle.api.dependencies.ResolverContainer`. This class provided methods for adding new resolvers at a particular position in the resolve chain.⁶ Ivy offers more complex structures for resolvers than simple chains. The resolver container does not offer particular support for this. But by manipulating its properties directly you could set up such other structures.

12.5.6 More about Ivy resolvers

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

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

If you declare a dependency like `junit:junit:3.8.2` 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 how this path should look like. Here are some examples:⁷

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

```
// Typical layout for an ivy repository (the organization is not split into subfolder)
someroot/[organisation]/[module]/[revision]/[type]/[artifact].[ext]
```

```
// Simple layout (the organization is not used, no nested folders.)
someroot/[artifact]-[revision].[ext]
```

To add any kind of repository (you can pretty easy write your own ones) you can do:

```
dependencies {
    classpathResolvers.add(new WebdavResolver() {
        name = 'gradleReleases'
        user = codehausUserName
        userPassword = codehausUserPassword
        addArtifactPattern(root + "[artifact]-[revision].[ext]")
    })
}
```

`WebdavResolver` is a class that implements an Ivy interface and is part of the *Gradle* distribution. 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.

⁶Ivy power users can set `dependencies.chainConfigurer = // some statements`. This closure is applied to the `ChainResolver` containing the other Ivy resolvers.

⁷At <http://ant.apache.org/ivy/history/latest-milestone/concept.html> you can learn more about ivy patterns.

Chapter 13

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 those tasks are executed in the order of their dependencies and are executed only once. Those tasks form an **Directed Acyclic Graph**. There are build tools that build up such a DAG as they execute there tasks. Gradle builds the complete DAG *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 DAG. Therefore they are strictly speaking *build configuration scripts*.

13.1 Build Phases

A Gradle build has three distinct phases.

Initialization Gradle supports single and multi-project builds. During the initialization phase, Gradle determines which project(s) are going to take part in the build. Also during this phase, Gradle creates Project objects for every project taking part in the build.

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

Execution A subset of the tasks, created and configured during the configuration phase, is executed. The subset is determined by the task name arguments passed to the gradle command and the current directory.

13.2 Settings File

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

The settings file gets executed during the initialization phase. A multiproject build must have a settings.gradle file in the root project of the multiproject hierarchy. It is required because in the settings.gradle file it is defined, which projects are taking part in the multi-project build (see chapter 14). For a single-project build, a settings.gradle file is optional. You might need it for example, to add libraries to your build script classpath (see chapter 15). Let's first do some introspection with a single project build:

```
----- settings.gradle -----  
println 'This is executed during the initialization phase.'
```

```
----- build.gradle -----  
println 'This is executed during the configuration phase.'  
  
createTask('test') {  
    println 'This is executed during the execution phase.'  
}
```

```
current dir: userguide/buildlifecycle  
>gradle test  
Modern compiler found.  
Recursive: true  
Buildfilename: gradlefile  
This is executed during the initialization phase.  
No build sources found.
```

```

:: loading settings :: url = jar:file:/Users/hans/IdeaProjects/gradle-release/gradle-core-RB0.1/build/
:: resolving dependencies :: org.gradle#build;SNAPSHOT
confs: [build]
++++ Starting build for primary task: test
++ Loading Project objects
++ Configuring Project objects
This is executed during the configuration phase.
Project=: evaluated.
++ Executing: test Recursive:true Startproject: :
Executing: :test
This is executed during the execution phase.
BUILD SUCCESSFUL
Total time: 1 seconds

```

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 [org.gradle.api.Settings](#).

13.3 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 14).

13.3.1 Project Locations

Multi-project builds are always represented by a tree with a single root. Each element in the tree represent a project. A project has a virtual and a physical path. The virtual path denotes the position of the project in the multi-project build tree. The project tree is created in the settings.gradle file. By default it is assumed that the location of the settings file is also the location of the root project. But you can redefine the location of the root project in the settings file.

13.3.2 Building the tree

In the settings file you can use a set of methods to build the project tree. Hierarchical and flat physical layouts get special support.

Hierarchical Layouts

```

dependencies {
    include 'project1', 'project2', 'project2/child1'
}

```

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

Flat Layouts

```

dependencies {
    includeFlat 'project1', 'project2'
}

```

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.

13.3.3 Modifying Elements of the Project Tree

The multi-project tree created in the settings file is made up of so called project descriptor object. You might modify those descriptor objects in the settings file at any time. To access such a descriptor object you can do:

```
dependencies {
    myDescriptor = descriptor('path_in_multi_project_tree')
    myOtherDescriptor = descriptor(new File('path_to_projectDir'))
}
```

Via this descriptor you can change the name and the directory of a project.

13.4 Initialization

How does Gradle know whether to do a single or multiproject build? In 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.¹ 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 not settings.gradle is found, it searches the parent directories for the existence of a settings.gradle 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 14). Via the `-u` command line option, you can tell Gradle not to look in the parent hierarchy for a settings.gradle file. The current project is then always build as a single project build. If the current project contains a settings.gradle file, the `-u` option has no meaning. Such a build is always executed as:

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

The auto search for a settings file does only work for multi-project builds with a physical hierarchical or flat layout. For a flat layout you must additionally obey to the naming convention described above. Gradle supports arbitrary physical layouts for a multiproject build. But for such arbitrary layouts you need to execute the build from the directory where the settings file is located. For how to run partial builds from the root see 14.4. In our next release we want to enable partial builds from subprojects by specifying the location of the settings file as a command line parameter.

Gradle creates Project objects for every project taking part in the build. For a single project build this is only one project. For a multi-project build these are the projects specified in Settings object (plus the root project). Each project object has by default a name equals to the name of its top level folder. Every project except the root project has a parent project and might have child projects.

13.5 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 equals 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 14.

¹Gradle supports partial multiproject builds (see chapter 14)

Chapter 14

Multi-project builds

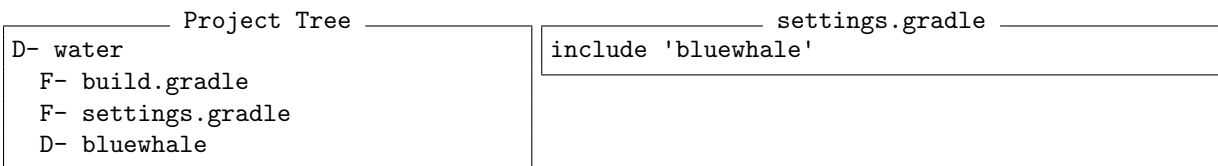
The powerful support for multi-project builds is one of Gradle's unique selling points. This topic is also the intellectually most challenging.

14.1 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.

14.1.1 Defining Common Behavior

We have the following project tree¹. This is a multi-project build with a root project `water` and a subproject `bluewhale`.



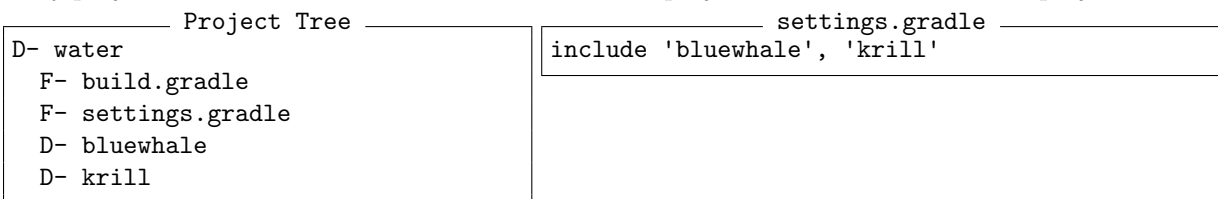
And where is the build script for the `bluewhale` project? In Gradle build scripts are optional. Obviously for a single project build, a project without a build script doesn't make much sense. For multi-project builds the situation is different. Let's look at the build script for the `water` project and execute it.:

```
water - build.gradle
Closure cl = { task -> println "I'm $task.project.name" }
createTask('hello', cl)
project(':bluewhale').createTask('hello', cl)
```

```
current dir: userguide/multi-project/firstExample/water
>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.



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

¹F means File and D means Directory

```

_____ water - build.gradle _____
allprojects {
    createTask('hello') { task -> println "I'm $task.project.name" }
}

```

```

_____ current dir: userguide/multiproject/addKrill/water _____
>gradle -q hello
I'm water
I'm bluewhale
I'm krill

```

Is this cool or is this cool? And how does this work? The Project API provides a property `allprojects` which returns a list with the current project and all its subprojects underneath it. If you call `allprojects` with a closure, the statements of the closure are delegated to the projects associated with `allprojects`. You could also do an iteration via `allprojects.each`, but that would be more verbose.

Other build systems use inheritance as the primary means for defining common behavior. We also offer inheritance for projects as you will see later. But Gradle uses **Configuration Injection** as the usual way of defining common behavior. We think it provides a very powerful and flexible way of configuring multiproject builds.

14.2 Subproject Configuration

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

14.2.1 Defining Common Behavior

```

_____ build.gradle _____
allprojects {
    createTask('hello') {task -> println "I'm $task.project.name" }
}
subprojects {
    hello.doLast {println "- I depend on water"}
}

```

```

_____ current dir: userguide/multiproject/useSubprojects/water _____
>gradle -q hello
I'm water
I'm bluewhale
- I depend on water
I'm krill
- I depend on water

```

14.2.2 Adding Specific Behavior

You can add specific behavior on top of the common behavior. Usually we put the project specific behavior in the build script of the project where we want to apply this specific behavior. But as we have already seen, we don't have to do it this way. We could add project specific behavior for the bluewhale project like this:

```

_____ water - build.gradle _____
allprojects {
    createTask('hello') {task -> println "I'm $task.project.name" }
}
subprojects {
    hello.doLast {println "- I depend on water"}
}
project(':bluewhale').hello.doLast {
    println "I'm the largest animal that has ever lived on this planet."
}

```

```

_____ current dir: userguide/multiproject/subProjectsAddFromTop/water _____
>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.

Project Tree

```
D- water
  F- build.gradle
  F- settings.gradle
  D- bluewhale
    F- build.gradle
  D- krill
    F- build.gradle
```

settings.gradle

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

```
bluewhale - build.gradle
hello.doLast { println "- I'm the largest animal that has ever lived on this planet." }
```

```
krill - build.gradle
hello.doLast {
  println "- The weight of my species in summer is twice as heavy as all human beings."
}
```

```
water - build.gradle
allprojects {
  createTask('hello') {task -> println "I'm $task.project.name" }
}
subprojects {
  hello.doLast {println "- I depend on water"}
}
```

current dir: userguide/multiproject/spreadSpecifics/water

```
>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.
```

14.2.3 Project Filtering

To show more of the power of *Configuration Injection*, let's add another project called tropicalFish and add more behavior to the build via the build script of the water project.

Filtering By Name

Project Tree

```
D- water
  F- build.gradle
  F- settings.gradle
  D- bluewhale
    F- build.gradle
  D- krill
    F- build.gradle
  D- tropicalFish
```

settings.gradle

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

```
water - build.gradle
allprojects {
  createTask('hello') {task -> println "I'm $task.project.name" }
}
subprojects {
  hello.doLast {println "- I depend on water"}
}
```

```

}
configureProjects(subprojects.findAll {it.name != 'tropicalFish'}) {
    hello.doLast {println '- I love to spend time in the arctic waters.'}
}

```

current dir: userguide/multiproject/addTropical/water

```

>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 `configureProjects` takes a list as an argument and applies the configuration to the projects in this list.

Filtering By Properties

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

Project Tree	settings.gradle
<pre> D- water F- build.gradle F- settings.gradle D- bluewhale F- build.gradle D- krill F- build.gradle D- tropicalFish F- build.gradle </pre>	<pre> include 'bluewhale', 'krill', 'tropicalFish' </pre>

bluewhale - build.gradle

```

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

```

krill - build.gradle

```

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

```

tropicalFish - build.gradle

```

arctic = false

```

water - build.gradle

```

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

```

current dir: userguide/multiproject/tropicalWithProperties/water

```

>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 `gradlefile` of the `water` project we use an `afterEvaluateListener`. 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 [14.6](#)

14.3 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.

```
current dir: current dir: userguide/multiproject/partialTasks/water/bluewhale
>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 Gradles 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 Gradles *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`.

```
bluewhale - build.gradle
arctic = true
hello.doLast { println "- I'm the largest animal that has ever lived on this planet." }

createTask('distanceToIceberg') {
    println '20 nautical miles'
}
```

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

createTask('distanceToIceberg') {
    println '5 nautical miles'
}
```

```
current dir: userguide/multiproject/partialTasks/water
>gradle -q distanceToIceberg
20 nautical miles
5 nautical miles
```

Here the output without the `-q` option

```
current dir: userguide/multiproject/partialTasks/water
>gradle distanceToIceberg
Modern compiler found.
Recursive: true
Buildfilename: build.gradle
No build sources found.
:: loading settings :: url = jar:file:/Users/hans/java/gradle-SNAPSHOT/lib/ivy-2.0.0.beta2_200803051655
```

```

:: resolving dependencies :: org.gradle#build;SNAPSHOT
   confs: [build]
++++ Starting build for primary task: distanceToIceberg
++ Loading Project objects
++ Configuring Project objects
Project=: evaluated.
Project=:bluewhale evaluated.
Project=:krill evaluated.
Project=:tropicalFish evaluated.
++ Executing: distanceToIceberg Recursive:true Startproject: :
Executing: :bluewhale:distanceToIceberg
20 nautical miles
Executing: :krill:distanceToIceberg
5 nautical miles

BUILD SUCCESSFUL

Total time: 1 seconds

```

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!

14.4 Running Tasks by there 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 14.5):

```

current dir: userguide/multiproject/partialTasks/water/tropicalFish
>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.

14.5 Project and Task Paths

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

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

Originally Gradle has used the `'/'` character as a natural path separator. With the introduction of directory tasks (see 4.3) this was no longer possible, as the name of the directory task contains the `'/'` character.

14.6 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:

14.6.1 Execution Dependencies

Dependencies and Execution Order

<pre>Project Tree D- messages F- settings.gradle D- consumer F- build.gradle D- producer F- build.gradle</pre>	<pre>settings.gradle include 'consumer', 'producer'</pre>
--	---

```
consumer - build.gradle
createTask('action') {
    println "Consuming message: " + System.getProperty('org.gradle.message')
}
```

```
producer - build.gradle
createTask('action') {
    println "Producing message:"
    System.setProperty('org.gradle.message', 'Watch the order of execution.')
}
```

```
current dir: userguide/multiproject/dependencies/firstMessages/messages
>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`.

<pre>Project Tree D- messages F- settings.gradle D- aProducer F- build.gradle D- consumer F- build.gradle</pre>	<pre>settings.gradle include 'consumer', 'aProducer'</pre>
---	--

```
aProducer - build.gradle
createTask('action') {
    println "Producing message:"
    System.setProperty('org.gradle.message', 'Watch the order of execution.')
}
```

```
consumer - build.gradle
createTask('action') {
    println "Consuming message: " + System.getProperty('org.gradle.message')
}
```

```
current dir: userguide/multiproject/dependencies/messagesHack/messages
>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.

```
current dir: userguide/multiproject/dependencies/messagesHack/messages/consumer
>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.

Declaring Dependencies

```

Project Tree
D- messages
  F- settings.gradle
  D- consumer
    F- build.gradle
  D- producer
    F- build.gradle
settings.gradle
include 'consumer', 'producer'

```

```

consumer - build.gradle
dependsOn(':producer')

createTask('action') {
    println "Consuming message: " + System.getProperty('org.gradle.message')
}

```

```

producer - build.gradle
createTask('action') {
    println "Producing message:"
    System.setProperty('org.gradle.message', 'Watch the order of execution.')
}

```

```

current dir: userguide/multiproject/dependencies/messagesHack/messages
>gradle -q action
Producing message:
Consuming message: Watch the order of execution.

```

```

current dir: userguide/multiproject/dependencies/messagesHack/messages/consumer
>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.

The Nature of Project Dependencies

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

```

consumer - build.gradle
dependsOn(':producer')

createTask('consume') {
    println "Consuming message: " + System.getProperty('org.gradle.message')
}

```

```

producer - build.gradle
createTask('produce') {
    println "Producing message:"
    System.setProperty('org.gradle.message', 'Watch the order of execution.')
}

```

```

current dir: userguide/multiproject/dependencies/messagesDifferentTaskNames/messages/consumer
>gradle -q consume
Consuming message: null

```

Uhps. Why does this not work? The `dependsOn` command is created for projects with a common lifecycle. Provided you have two Java projects were one depends on the other. If you trigger a compile for 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:

```

consumer - build.gradle
createTask('consume', dependsOn: ':producer:produce') {
    println "Consuming message: " + System.getProperty('org.gradle.message')
}

```

```

----- producer - build.gradle -----
createTask('produce') {
    println "Producing message:"
    System.setProperty('org.gradle.message', 'Watch the order of execution.')
}

```

```

current dir: userguide/multiproject/dependencies/messagesTaskDependencies/messages/consumer
>gradle -q consume
Producing message:
Consuming message: Watch the order of execution.

```

14.6.2 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.

```

----- consumer - build.gradle -----
key = 'unknown'
if (project(':producer').hasProperty('key')) {
    key = project(':producer').key
}
createTask('consume', dependsOn: ':producer:produce') {
    println "Consuming message from key '$key': " + System.getProperty(key)
}

```

```

----- producer - build.gradle -----
key = 'org.gradle.message'

createTask('produce') {
    println "Producing message:"
    System.setProperty(key, 'Watch the order of execution.')
}

```

```

----- current dir: consumer -----
>gradle -q consume
Producing message:
Consuming message from key 'unknown': null

```

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

```

----- consumer - build.gradle -----
evaluationDependsOn(':producer')

key = 'unknown'
if (project(':producer').hasProperty('key')) {
    key = project(':producer').key
}
createTask('consume', dependsOn: ':producer:produce') {
    println "Consuming message from key '$key': " + System.getProperty(key)
}

```

```

current dir: userguide/multiproject/dependencies/messagesConfigDependencies/messages/consumer
>gradle -q consume
Producing message:
Consuming message from key 'org.gradle.message': Watch the order of execution.

```

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

```

----- consumer - build.gradle -----
createTask('consume', dependsOn: ':producer:produce') {
    String key = project(':producer').key
    println "Consuming message from key '$key': " + System.getProperty(key)
}

```

```
current dir: userguide/multiproject/dependencies/messagesConfigDependencies/messages/consumer
>gradle -q consume
Producing message:
Consuming message from key 'org.gradle.message': Watch the order of execution.
```

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

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

14.6.3 Real Life examples

Gradles 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.² For the example we use only one build script and do *cross project configuration*.

```
Project Tree
D- webDist
  F- settings.gradle
  F- build.gradle
  D- date
    F- src/main/java/org/gradle/sample/DateServlet.java
  D- hello
    F- src/main/java/org/gradle/sample/HelloServlet.java
```

```
settings.gradle
include 'date', 'hello'
```

```
webDist - build.gradle
dependsOnChildren()

allprojects {
    usePlugin('java')
    sourceCompatibility = 1.5
    targetCompatibility = 1.5
    group = 'org.gradle.sample'
    version = '1.0'
}

subprojects {
    usePlugin('war')
    dependencies {
        addMavenRepo()
        compile "javax.servlet:servlet-api:2.5"
    }
}

createTask('explodedDist', dependsOn: 'libs') {
    File explodedDist = mkdir(buildDir, 'explodedDist')
    subprojects.each {project ->
        project.libs.archiveTasks.each {archiveTask ->
            ant.copy(file: archiveTask.archivePath, todir: explodedDist)
        }
    }
}
```

²The 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.

We have an interesting set of dependencies. Obviously the `date` and `hello` task have a *configuration* dependency on `webDist`, as all the build logic for the webapp projects is injected by `webDist`. The *execution* dependency is in the other direction, as `webDist` depends on the build artifacts of `date` and `hello`. There is even a third dependency. `webDist` has a *configuration* dependency on `date` and `hello` because it needs to know the `archivePath`. But it asks for this information at *execution time*. Therefore we have no circular dependency.

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

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

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

14.7 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 12.2.5, Gradle offers project dependencies for this.

```

Project Tree
D- java
  D- api
    F- src/main/java/org/gradle/sample/api/Person.java
    F- src/main/java/org/gradle/sample/api/PersonImpl.java
  D- services
    D- personService
      F- src/main/java/org/gradle/sample/services/PersonService.java
      F- src/main/test/org/gradle/sample/services/PersonServiceTest.java
  D- shared
    F- src/main/java/org/gradle/sample/shared/Helper.java
  F- settings.gradle
  F- build.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`.³

```

settings.gradle
include 'api', 'shared', 'services:personService'

```

```

userguide/multiproject/dependencies/java - build.gradle
subprojects {
    usePlugin('java')
    sourceCompatibility = 1.5
    targetCompatibility = 1.5
    group = 'org.gradle.sample'
    version = '1.0'
}

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

project(':services:personService') {
    dependencies {
        compile project(':shared'), project(':api')
        testCompile "junit:junit:3.8.2"
    }
}

```

³`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.

All the build logic is in the `build.gradle` of the root project.⁴ A *lib* dependency is a special form of an execution dependency. It causes the other project to be build first and adds the jar with the classes of the other project to the classpath. It also add the dependencies of the other project to the classpath. So you can enter the `api` folder and trigger a `gradle compile`. First `shared` is build and then `api` is build. Project dependencies enable partial multi-project builds.

If you come from Maven land you might be perfectly happy with this. If you come from Ivy land, you might expect some more fine grained control. Gradle offers this to you:

```
userguide/multiproject/dependencies/javaWithCustomConf - build.gradle
subprojects {
    usePlugin('java')
    sourceCompatibility = 1.5
    targetCompatibility = 1.5
    group = 'org.gradle.sample'
    version = '1.0'
}

project(':api') {
    dependencies {
        addConfiguration('spi')
        compile project(':shared')
    }
    libs.jar('api-spi') {
        configurations('spi')
        fileSet() {
            include('org/gradle/sample/api/')
        }
    }
}

project(':services:personService') {
    dependencies {
        compile project(':shared'), new ProjectDependency(project(':api'), 'spi')
        testCompile "junit:junit:3.8.2", project(':api')
    }
}
```

The Java plugin adds per default a jar to your project libraries which contains all the classes. In this example we create an *additional* library containing only the interfaces of the `api` project. We assign this library to a new *dependency configuration*. For the person service we declare that the project should be compiled only against the `api` interfaces but tested with all classes from `api`.

14.8 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 userguide we might write more about this.

Method inheritance might be interesting to use as Gradles *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.

14.9 Summary

Writing this chapter was pretty exhausting and reading it might have a similar effect. Our final message for this chapter is that multi-project builds with Gradle are usually *not* difficult. There are six elements you need to remember: `allproject`, `subprojects`, `dependsOn`, `childrenDependOnMe`, `dependOnChildren` and `project lib dependencies`.⁵ With those elements, and keeping in mind that Gradle has a distinct configuration and execution

⁴We do this here, as it makes the layout a bit easier. We usually put the project specific stuff into the buildscript of the respective projects.

⁵So we are well in the range of the **7 plus 2 Rule** :)

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.

Chapter 15

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.¹ Gradle makes this very easy. Just drop your classes in a certain folder and Gradle automatically compiles them and puts them in the classpath of your build script.

15.1 Build Sources

If you run Gradle, it checks for the existence of a folder called `buildSrc`. Just put your build source code in this folder and stick to the layout convention for a Java/Groovy project (see Table 9.1). Gradle then automatically compiles and tests this code and puts it in the classpath of your build script. You don't need to provide any further instruction. For multi-project builds there can be only one `buildSrc` directory which has to be in the root project.

This is probably good enough for most of the cases. If you need more flexibility, you can provide a `build.gradle` and a `settings.gradle` file in the `buildSrc` folder. If you like, you can even have a multi-project build in there.

15.2 External dependencies

If your build script needs external libraries you can declare them in the `settings.gradle` file.

```
dependencies("commons-math:commons-math:1.1:jar")
```

You can pass any of the dependencies described in section ?? (except project dependencies). There is *no* need to provide a *dependency configuration* (e.g. `compile`). For multi-project builds dependencies declared in the `settings.gradle` file of the root project, are available to all build scripts of the sub-projects.

15.3 Ant Optional Dependencies

For reasons we don't fully understand yet, external dependencies are not picked up by Ant's optional tasks. But you can easily do it in another way.²

```
dependencies {
    addConfiguration('ftpAntTask')
    clientModule(['ftpAntTask'], ":ant-commons-net:1.7.0") {
        clientModule(":commons-net:1.4.1") {
            dependencies(":oro:2.0.8:jar")
        }
    }
}
createTask('ftp') {
    ant {
        taskdef(name: 'ftp',
```

¹Which might range from a single class to something very complex)

²In fact, we think this is anyway the nicer solution. Only if your buildscript and Ant's optional task need the *same* library you would have to define it two times. In such a case it would be nice, if Ant's optional task would automatically pickup the classpath defined in the `gradleSettings`.

```
        classname: 'org.apache.tools.ant.taskdefs.optional.net.FTP',
        classpath: dependencies.antpath('ftpAntTask'))
    ftp(server: "ftp.apache.org", userid: "anonymous", password: "me@myorg.com") {
        fileset(dir: "htdocs/manual")
    }
}
```

15.4 Summary

Gradle offers you a variety of ways of organizing your build logic. You can choose what is right for your domain and find the right balance between unnecessary indirections, and avoiding redundancy and a hard to maintain code base. It is our experience that even very complex custom build logic is rarely shared between different builds. Other build tools enforce a separation of this build logic into a separate project. Gradle spares you this unnecessary overhead and indirection.

Chapter 16

The Gradle Wrapper

Gradle is a new tool. You can't expect it to be installed on machines beyond your sphere of influence. An example are continuous integration server where Gradle is not installed and where you have no admin rights for the machine. Or what if you provide an open source project and you want to make it as easy as possible for your users to build it?

There is a simple and good news. Gradle provides a solution for this. It ships with a *Wrapper* task.¹² You can create such a task in your build script.

```
createTask('wrapper', type: Wrapper).configure {
    gradleVersion = '0.1'
}
```

You usually explicitly execute this task (for example after a switch to a new version of Gradle). After such an execution you find the following new or updated files in your project folder (if the default configuration is used).

```
project-root
- gradle-wrapper.jar
- gradlew.exe
- gradlew.bat
```

All these files should be submitted to your version control system. The `gradlew` commands can be used *exactly* the same way as the `gradle` commands.

16.1 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 the download URL of the wrapper distribution. You can also specify where the wrapper 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. 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.

For the details on how to configure the wrapper, see [Wrapper API](#)

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

16.2 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`.

¹If you download the gradle source distribution or check out Gradle from SVN, you can build Gradle via the gradle wrapper.

²Gradle itself is continuously built by Bamboo and Teamcity via this wrapper. See <http://gradle.org/ci-server.html>

16.3 Environment variable

Some rather exotic use cases might occur when working with the Gradle Wrapper. For example the continuous integration server goes down during unzipping the Gradle distribution. As the distribution directory exists `gradlew` delegates to it but the distribution is corrupt. Or the zip-distribution was not properly downloaded. When you have no admin right on the continuous integration server to remove the corrupt files, Gradle offers a solution via environment variables.

Variable Name	Meaning
GRADLE_WRAPPER_ALWAYS_UNPACK	If set to <code>true</code> , the distribution directory gets always deleted when <code>gradlew</code> is run and the distribution zip is freshly unpacked. If the zip is not there, Gradle tries to download it.
GRADLE_WRAPPER_ALWAYS_DOWNLOAD	If set to <code>true</code> , the distribution directory and the distribution zip gets always deleted when <code>gradlew</code> is run and the distribution zip is freshly downloaded.

Appendix A

Potential Traps

A.1 Groovy Script Variables

For Gradle users it is important to understand how Groovy deals with script variables. Groovy has two types of script variables. One with a local scope and one with a script wide scope.

```
String localScope1 = 'localScope1'
def localScope2 = 'localScope2'
scriptScope = 'scriptScope'

println localScope1
println localScope2
println scriptScope

closure = {
    println localScope1
    println localScope2
    println scriptScope
}

def method() {
    try {localScope1} catch(MissingPropertyException e) {println 'localScope1NotAvailable' }
    try {localScope2} catch(MissingPropertyException e) {println 'localScope2NotAvailable' }
    println scriptScope
}

closure.call()
method()
```

```
>groovy scope.groovy
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.¹

A.2 Configuration and Execution Phase

It is important to keep in mind that Gradle has a distinct configuration and execution phase (see chapter 13).

¹One of those discussions can be found here: <http://www.nabble.com/script-scoping-question-td16034724.html>

```
classesDir = new File('build/classes')
classesDir.mkdirs()
createTask('clean') {
    ant.delete(dir: 'build')
}
createTask('compile', dependsOn: 'clean') {
    if (!classesDir.isDirectory()) {
        println 'The class directory does not exists. I can not operate'
        // do something
    }
    // do something
}
```

```
>gradle -q compile
The class directory does not exists. 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.

Appendix B

Existing IDE Support and how to cope without it

B.1 IntelliJ

Gradle has been mainly developed with Idea IntelliJ and its very good Groovy plugin. Gradle's build script¹ 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 `build.gradle`s get special treatment by IntelliJ and you will be able to define a specific classpath for them.

B.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 userguide we can hopefully write more about this.

B.3 Using Gradle without IDE support

What we can do for you is to spare you typing things like `throw new org.gradle.api.tasks.StopExecutionException()` and just type `throw new StopExecutionException()` instead. We do this by automatically adding a set of import statements to the gradle scripts before Gradle executes them. This set is defined by a properties file `gradle-imports` in the gradle distribution. It has the following content.

```
import org.gradle.*
import org.gradle.util.*
import org.gradle.api.*
import org.gradle.api.dependencies.*
import org.gradle.api.initialization.*
import org.gradle.api.plugins.*
import org.gradle.api.tasks.*
import org.gradle.api.tasks.bundling.*
import org.gradle.api.tasks.compile.*
import org.gradle.api.tasks.javadoc.*
import org.gradle.api.tasks.testing.*
import org.gradle.api.tasks.util.*
import org.gradle.api.tasks.wrapper.*
```

You can define a project specific set of imports to be added to your build scripts. Just place a file called `gradle-imports` in your root project directory. If you start Gradle with the `-I` option, the imports defined in the Gradle distribution are disabled. The imports defined in your project directory are always used.

¹Gradle is build with Gradle

Appendix C

Command line

Option(-)	Long Option(-)	Meaning
-?, -h	-	Shows this help message
B	bootstrap-debug	Specify a text to be logged at the beginning (used internally by Gradle's bootstrap class)
D	systemprop	Sets a system property of the JVM (e.g. -Dmyprop=myvalue).
I	noImports	Disable usage of default imports for build script files.
P	projectprop	Sets a project property of the root project (e.g. -Pmyprop=myvalue).
S	-	Don't trigger a System.exit(0) for normal termination. Used internally by Gradle's unit tests.
b	buildfile	Specifies the build file name (also for subprojects). Defaults to build.gradle.
c	settingsfile	Specifies the settings file name. Defaults to settings.gradle.
d	debug	Log in debug mode (includes normal stacktrace).
e	embedded	Specify an embedded build script.
f	full-stacktrace	Print out the full (very verbose) stacktrace for any exceptions.
g	gradle-user-home	Specifies the gradle user home dir.
i	ivy-quiet	Set Ivy log level to quiet.
j	ivy-debug	Set Ivy log level to debug (very verbose).
l	plugin-properties-file	Specifies the plugin properties file.
p	project-dir	Specifies the start dir for Gradle. Defaults to current dir.
q	quiet	Log errors only.
r	rebuild-cache	Rebuild the cache of compiled build scripts.
s	stacktrace	Print out the stacktrace also for user exceptions (e.g. compile error).
t	tasks	Show list of all available tasks and there dependencies.
u	no-search-upwards	Don't search in parent folders for a settings.gradle file.
v	version	Prints version info.
x	cache-off	No caching of compiled build scripts.

The same information is printed to the console when you execute `gradle -h`.