jahe / **gradle-cheatsheet.gradle**

Last active 8 days ago • Report gist

Gradle Cheatsheet

gradle-cheatsheet.gradle

```
// imports a couple of java tasks
apply plugin: "java"

// List available tasks in the shell
> gradle tasks

// A Closure that configures the sourceSets Task
// Sets the main folder as Source folder (where the compiler is looking up the .java files)
sourceSets {
  main.java.srcDir "src/main"
}

// This can also be written as a function -> srcDir is a method (Syntax sugar of the Groovy language)
souceSets {
  main.java.srcDir("src/main")
}

// Or
sourceSets.main.java.srcDir "src/main"

// Or
sourceSets {
  main {
    java {
      srcDir "src/main"
    }
  }
}

// Or setting the variable directly as a typical groovy enumerational style
sourceSets {
  main.java.srcDirs = ["src/main"]
}

// Compile and Test the Java Project into a build directory and package it in a .jar file
> gradle build

// Configure the jar task to insert the Main Class to the resulting MANIFEST.MF
jar {
  manifest.attributes("Main-Class", "de.example.main.Application")
}

// Or without the parentheses
jar {
  manifest.attributes "Main-Class": "de.example.main.Application"
}

// Configure Dependencies of the Java Project (gradle supports maven and ivy repos by default)
// by first defining the repos with maven in another closure
repositories {
  maven {
    url "mvn-repo-xy.de"
  }
}

// Or using the mavenCentral method that is built in gradle by default
repositories {
  mavenCentral()
}
```

```

// Or using the mavenLocal method which is using the local maven cache of your own server with your own archetypes
repositories {
  mavenLocal()
}

// Now we can set the dependencies by configuring the dependencies closure
// where compile is a configuration not a method that compiles the dependency or so
// It puts the dependency in the classpath of the Java Application
// <groupId>:<artifactId>:<version>
dependencies {
  compile "org.apache.commons:commons-lang3:3.3.2"
}

// Or
dependencies {
  compile group: "org.apache.commons", name: "commons-lang3", version: "3.3.2"
}

// To see which configuration there are for a task e.g. for dependencies
> gradle dependencies

// Build the project including downloading the dependency from the maven central repository
> gradle build
> ...
> BUILD SUCCESSFUL
// The Build was successful because gradle puts the dependency in the classpath by compiling it
// but the packaged .jar file doesn't have this dependency in its classpath.
// It has only the classes from the src/main folder in the classpath which is specified in the sourceSets
// so it will throw an NoClassDefFoundError Exception by running the .jar

// Thats ok, because you not always want to have all dependencies in the .jar
// To have the dependency in the .jar's classpath we have to configure the jar closure
// by using the file collection of the "compile ..." statement to insert it in the .jar's classpath
// To get the file collection we have to use the method "from" with the "configurations" property of the project
// With "configurations.compile" we get a file collection of .jar's which are stated in the dependencies closure
// in the form of maven artifacts (which are just links to .jar files)
// With the groovy method "collect" (which is available because the "compile" file collection implements
// the groovy Collections Interface) we can transform/replace the collection by specifying a closure
// which unzips the dependencies .jar and copies the classes of it in the build folder
jar {
  from configurations.compile.collect {
    entry -> zipTree(entry)
  }
}

// Or with syntactic sugar
jar {
  from configurations.compile.collect {
    entry -> zipTree entry
  }
}

// And with more syntactic sugar
// (where "it" is like "this" in gradle and "this" is the entry iterating over by the for loop)
jar {
  from configurations.compile.collect {
    zipTree it
  }
}

// These configurations come from the java plugin
// We can also specify our own configurations
// where myConfig is a simple empty file collection
configurations {
  myConfig
}

// Now we can say that "myConfig" is going to take the dependency
dependencies {
  myConfig "org.apache.commons:commons-lang3:3.3.2"
}

// And say this because "myConfig" is now the file collection with the .jar from the maven dependency
jar {
  from configurations.myConfig.collect {
    zipTree it
  }
}

// What we also can do is: myConfig <taskname>
// to put the output of the task in the myConfig configuration
artifacts {
  myConfig jar
}

// Strings in Double Quotes are GStrings which have templating functionalities
apply plugin "java $variable"

// Strings in Single Quotes are simple Java Strings
apply plugin 'java'

// Create / Declare a Task
// A Task is a first-class object
task hello

> gradle tasks
...
Other tasks
-----------
hello
...

// Create a Task with Metadata
task hello(group: 'greeting', description: 'Greets you.')

> gradle tasks
...
Greeting Tasks
--------------
hello - Greets you.

// Or with syntax sugar
task hello {
  group 'greeting'
  description 'Greets you.'
}

// Do the Greeting in the task
task hello {
  group 'greeting'
  description 'Greets you.'

  doLast {
    println 'Hello!'
  }
}

> gradle hello
...
:hello
Hello!
...

// A Task has properties like "group" and "description" and a queue of actions that is supposed to execute
// and content of the closure "doLast" is one of them.
// doLast is only the method of the task that appends the "println" action to the end of the action queue
// so that we can add more than just one "doLast" closure to the task

// Put an action to the beginning of the action queue
```

```
task hello {
  doLast { println 'Hello!' }
  doFirst { 'Hey I know this guy' }
}

// Put an action to the action queue outside the task
hello << { println 'I was appended using <<' }

// Or with syntax sugar:
hello.doLast { println 'I was appended using .doLast' }

// We could also append an action with the left shift << directly after the task closure
task hello {
  doLast { println 'Hello!' }
  doFirst { 'Hey I know this guy' }
} << {
  println 'I was appended directly after the closure' }
}

// We can also put an action in the task directly but that is different
// as the action queue is executed during the execution phase
// and the "println 'hello'" is executed during the configuration phase
// which is also called in "gradle tasks" command even though we didn't
// executed the "hello" task and it will appear on every command not just
// on the "gradle tasks" command because there always be a configuration phase
// where the tasks are being configured/prepared for execution
task hello {
  println 'Hello from the configuration phase'

  doLast { println 'Hello!' }
  doFirst { 'Hey I know this guy' }
}

> gradle hello
...
Hello from the configuration phase
:hello
Hey I know this guy
Hello!
...

> gradle tasks
Hello from the configuration phase
:tasks
...

// It is possible to set extra properties in the configuration phase
// that are being evaluated in the action queues actions
// in the doLast we use the GStrings templating options
task hello {
  println 'Hello from the configuration phase'
  ext.greeting = 'Hey, how\'s it going?'

  doLast { println "Greeting: $greeting" }
}

// Now a useful task: Run a .jar
// It is of the type "Exec". It executes a command line process
// => java -jar thejar.jar "hello" "world"
//
// The second argument is written as a GString Template.
// $jar accesses a variable which is the task "jar"
// and .archivePath is the property of that task
// where the .jar is constructed
//
// When the runJar task is executed we have to provide that
// the .jar is already created. Therefore we annotate the task
// with a "dependsOn" keyword which will run the specified task
// first before the actual task is being executed.
//
```

```
// When a task depends on another task it has to be declared
// before the actual task otherwise it'll break/don't find the specified task
task runJar(type: Exec, dependsOn: jar) {
  executable 'java'
  args '-jar', "$jar.archivePath", 'Hello', 'World'
}

> gradle runJar
...
:runJar
Hello
World
...

// Or with syntax sugar we can set the type and the depends on within the task
task runJar {
  type Exec
  dependsOn jar

  executable 'java'
  args '-jar', "$jar.archivePath", 'Hello', 'World'
}

// Run the java program without the jar packaging directly from the .class files
// JavaExec is a subclass of Exec which executes .class files without having a .jar file
// The classes task assembles/creates .class files
// from the specified sourceSets.main property (e.g. 'src/main')
task run(type: JavaExec, dependsOn: classes) {
  main 'gradledemo.Main'
  classpath sourceSets.main.runtimeClasspath
  args 'Hello', 'World'
}

> gradle run
...
:classes
:run
Hello
World
...

// Only execute a task in specific conditions
// Therefore the onlyIf closure returns true or false
// resulting from the expression inside the closure.
// (In Groovy the last statement of a closure is the return value)
// When onlyIf evaluates to false the log shows SKIPPED beside the taskname
// onlyIf is a method of the task and can be called outside of the configuration closure
task hello {
  onlyIf { false }
} << {
  println 'Hello!'
}

> gradle hello
...
:hello SKIPPED
...

// Or call the onlyIf(closure) method of the task
task hello << {
  println 'Hello!'
}
hello.onlyIf { false }

// Enable/Disable a task
// If a task is disabled it'll be always SKIPPED even if onlyIf returns true
task hello {
  doLast { println 'Hello!' }
}

```

```
hello.enabled = false

> gradle hello
...
:hello SKIPPED
...

// Write a Greeting to a file within a doLast closure
task writeGreeting << {
  file('greeting.txt').text = 'Hello guys!'
}

// Write the greeting in the file only if the file + the content doesn't match
task writeGreeting {
  onlyIf { !file('greeting.txt').text.equals('Hello guys!') }
} << {
  file('greeting.txt').text = 'Hello guys!'
}

// A task can have inputs and outputs (properties)
// When the outputs specified in the task are already there
// and have the same content (which is done by building a checksum of
// the files content)
// Gradle says "UP-TO-DATE" next to the task name
// This
task writeGreeting {
  outputs.file file('greeting.txt')
} << {
  file('greeting.txt').text = 'Hello guys!'
}

> gradle writeGreeting
:writeGreeting UP-TO-DATE

// Passing parameters into the build script
// with system properties
> gradle -D<property>=<value>

// e.g. sets the "custom.config" property to "my-config.properties"
> gradle -Dcustom.config=my-config.properties

// Set the Loglevel to INFO to get a couple more infos while executing
> gradle --info

// Or
> gradle -i

// Set the Loglevel to DEBUG to see stacktraces etc.
> gradle --debug

// Or
> gradle -d

// Evaluate the build script for errors and run it, but do not execute a task
> gradle --dry-run

// Or
> gradle -m

// Run the build script in quite mode which only prints out error messages
> gradle --quite

// Or
> gradle -q

// Run Gradle with the Gradle GUI
> gradle --gui

// Show an abbreviated (groovy internal method calls removed) stack trace when an exception is thrown in the build script
// Nice for debugging a broken build
// (There is also a --full-stacktrace or -S option for printing internal groovy methods as well)
> gradle --stacktrace

// Or
> gradle -s

// Show all properties of the builds project object
// The project object represents the structure and state of the current build
> gradle properties

// There are 3 lifecycles in gradle script execution:
// 1. Initialization
// 2. Configuration
// 3. Execution

// There are configuration and execution closures in a task
// Both of them are additive
task hello
hello << { println 'hello ' }
hello << { println 'world' }
hello { print 'configuring ' }
hello { println 'hello task' }

// The Configuration blocks/closures are used for setting up variables
// and data structures that will be needed by the tasks action
// It turns the tasks into rich object models populated with information
// about the build (rather than a strict sequence of build actions)

// Tasks are Objects with methods and properties
// Their default type is "DefaultTask" which only provides the interface
// to the Gradle project model.

// ------------------------
// METHODS of "DefaultTask"
// ------------------------

// dependsOn(task)
// Declare that world depends on hello
task world {
  dependsOn hello
}

// Or with syntax sugar:
task world {
  dependsOn << hello
}

// Or with syntax sugar using single quotes (which are optional)
task world {
  dependsOn 'hello'
}

// Or explicitly call the "dependsOn" method on the task method
task world
world.dependsOn hello

// Or with a shortcut
task world(dependsOn: hello)

// Declaring multiple dependencies
task world {
  dependsOn << prepareHelloWorld
  dependsOn << hello
}

// Or pass dependencies as a variable-length list
task world {
  dependsOn prepareHelloWorld, hello
}

```

```
// Or explicitly call the method on the task object
task world
world.dependsOn prepareHelloWorld, hello

// A shortcut for dependencies only
// Note the Groovy list syntax
task world(dependsOn: [ prepareHelloWorld, hello ])

// doFirst is also a method on the task object which accepts a closure
// that will be put to the beginning of the tasks action queue
task world
world.doFirst {
  println ' world'
}

// doFirst can also be called from within the configuration block/closure
task world {
  doFirst { println 'world' }
}

// Execute a task only if a cli argument is set to true
loadTestData.onlyIf {
  System.properties['load.data'] == 'true'
}

> gradle loadTestData
> :loadTestData SKIPPED

> gradle -Dload.data=true loadTestData
> :loadTestData
> load test data

// Set/Get the didWork property of a task
// It is part of the properties of the "DefaultTask"
// It indicates the success or failure of its actions
// Lets check the "didWork" property of the "compileJava"
// task to decide whether to send a success E-Mail or not

task emailMe(dependsOn: compleJava) << {
  if (tasks.compileJava.didWork) {
    println 'SEND EMAIL ANNOUNCING SUCCESS'
  }
}

// Create an initial project structure
gradle init --type java-library

// 4 types of dependencies:
// compile: Dependencies to compile the sources. The smallest set of dependencies (works as the base for all other types)
// runtime: Dependencies to run the application.
// testCompile: Dependencies to compile the tests. (includes compile dependencies)
// testRuntime: Dependencies to run the tests. (includes testCompile dependencies)

// Ignore test failures so that the overall build doesn't fail (i.e. in prototyping situations)
test {
  ignoreFailues = true
}

// Configure JAR Task with specific Manifest values
jar {
  manifest {
    attributes (  "Implementation-Title": "<title>",
                  "Implementation-Version": version,
                  "Main-Class": "de.example.HelloWorld" )
  }
}

// Add a code checker to the build (reports in "build/reports/pmd")
apply plugin: 'pmd'

```

```
// "gradle eclipse" generates .project + .classpath with correct dependencies as "Referenced Libraries"
apply plugin: 'eclipse'

// Set files/dirs to compile und the output directory (when the maven/gradle project structure isn't used)
sourceSets {
  main {
    java {
      srcDir = ['src']
      output.classesDir = ['bin']
    }
  }
}

// Multiproject has build.gradle + settings.gradle
// settings.gradle:
include 'gui', 'model', 'dao'

// build.gradle:
subproject {
  apply plugin: 'java'
  repositories {
    jcenter()
  }
}
```

hkuadithya commented on Feb 22 2017

Thank you for this cheet sheet.
Could you please add configuration for publishing the JAR sources too.
For instance, I want to publish JARs but I want to publish JAR sources too which
will be convenient for consumers of my JAR to review the JAR code.

jahe commented on Feb 23 2017 Owner

@hkuadithya: I'm glad to hear that it helped you.
I'll look into it and see what I can do to add this to the cheat sheet.
PRs welcome btw.