Building a working `./gradlew build`

A collection of best practices, pain points, and tricks
that I hard to learn the hard way

Patrick S. Rhomberg,
a.k.a. GitHub user `PurelyApplied`

June 28, 2019

Contents

List of Snippets

1 Introduction and disclaimer

Pretend this is a dev blog. I intend, in fact, to upload this to a personal dev blog. Remember read this with that mentality. Despite the nice presentation that comes from my love of LaTeX, this is really just a haphazard collection of things I subjectively consider important for someone trying to ramp up or stabilize their Gradle build.

Some things are initially presented in the "easy to think this way that is a little bit wrong." I footnote these with the correction. This document is meant to be educational, not necessarily as a concise reference, and the path to understanding is rarely direct. I know it can seem misleading, particularly to people who are technically inclined, but here we are.

I learned Gradle the proverbial Hard Way while revamping a legacy build into something parallelizable, correct, and consistent.[1] It expects a passing familarity but not a comprehensive understanding. This is for people trying to understand their Gradle build, not necessarily for someone trying to write a new one.

2 Groovy crash course

Groovy feels like Java's side-hustle moonlighting as a scripting language. It is much more permissive than Java is, which can be freeing at times and can feel like more than enough rope at others. If you want an actual syntax-and-such crash course, I encourage you to read the docs.[2] Alternatively, I'm a personal fan of Learn X in Y Minutes.[3]

The things I mention here have more to do with what you might find confusing or counter-intuitive.

2.1 Functions don't require parentheses

Something that can be initially confusing is the fact that Groovy can accept function parameters without parentheses. See Snippet 1 for an explcit example. This is true even for functions that accept more than one argument, although in this case parentheses often improve readability. Also, nesting function calls *does* require parentheses.

```
println "Hello world!"
// is equivalent to
println("Hello world!")


def add(int x, int y) {
  return x + y
}

def z = add(1, 2)
// is equivalent to
def z = add 1, 2
```

Snippet 1: Parentheses are often optional in Groovy.

As we'll see in a moment, there is a lot of wiggle room when it comes to parentheses and how arguments are passed to Groovy functions. This is particularly true when it comes to closures. "But what are closures?" I hear you cry...

[1] Or at least, *more* parallelizable, correct, and consistent.
[2] `http://groovy-lang.org/documentation.html`
[3] `https://learnxinyminutes.com/docs/groovy/`

2.2 Closures

Closures are the Big Hotness in Groovy. You can start off by thinking of a closure as a super-lambda.[4] Closures are ubiquitous in Gradle because they're a wonderful way to specify configuration as a parameter, in a reasonably readable and intuitive way.

Closures look like (and really *are*) an arbitrary code block. They're offset by squiggly-braces, like any function definition or `for` loop block.

Closures often have an implicit parameter of `it`. See Snippet 2 for a trivial example.

```
def greeting1 = { "Hello, $it!" }
def greeting2 = { it -> "Hello, $it!" }
def greeting3 = { name -> "Hello, $name!" }
assert greeting1("Patrick") == "Hello, Patrick!"
assert greeting2("Patrick") == "Hello, Patrick!"
assert greeting3("Patrick") == "Hello, Patrick!"
```

Snippet 2: Closure can explicitly define parameters, or otherwise have the implicit parameter `it`.

2.3 More than super-lambdas

The real power of closures comes from *delegate injection*. Unlike a Java lambda, closures are [TODO WORDS]

```
class MyStringContainer {
  private myString

  MyStringContainer(String s) {
    myString = s
  }

  String getMyString(){
    println "In getter"
    return myString
  }

  String setMyString(String s){
    println "In setter"
    myString = s
  }


  String plus(String s){
    return myString + s
  }
}
```

Snippet 3: We will use this simple class in many of the subsequent examples.

[4] While this is absolutely incorrect, it's a reasonable starting point that we will correct in a moment. As we'll see, closures are first-class objects and have the advantage of being able to be configured after instantiation in ways that Java lambdas do not. See `http://groovy-lang.org/closures.html#_groovy_closures_vs_lambda_expressions` for details.

Consider the following function which takes a `MyStringContainer` and a closure.

```
String myFunc(MyStringContainer container,
              Closure someClosure){
  cl.delegate = container
  return someClosure()
}

println myFunc(new MyStringContainer("Hello"), {
  plus " world!"
})
```

Snippet 4: boo

This snippet will print `Hello world!` [TODO explain why]

2.4 Syntactic sugar

2.4.1 Setters are getters

Setters and getters are boring. Groovy doesn't make you use them. Consider Snippet 5

The following are equivalent.

```
// In the Java-style
MyStringContainer x = new MyStringContainer("foo")
println(x.getMyString())
x.setMyString("bar")
println(x.getMyString())

// Equivalent function calls in the Groovy style
def x = new MyStringContainer("foo")
println x.myString
x.myString = "bar"
println x.myString
```

Snippet 5: Groovy uses setters and getters but adopts a more direct syntax.

Note that if you run this locally, you *do* get the `"In setter"` and `"In getter"` prints in both cases.

2.4.2 Playing fast and loose with parentheses return values

Let us consider again the snippet from Section 2.3. In Snippet 6, we see two bits of syntactic sugar.

Groovy does not require an explicit `return`, and in its absence, will return the value of the last line called. Also, Groovy will allow you to close the paren before the last argument of a function if that final argument is a closure. As such, the following snippet is equivalent to the above.

2.4.3 Corollary: What seems like keywords are actually function calls

Consider the code presented in Snippet 7, pulled from Apache Geode's build. In this snippet, we declare a task of type `Jar` and configure it.

The following are equivalent:

```
String myFunc(MyStringContainer container,
              Closure someClosure){
  cl.delegate = container
  return someClosure()
}

println myFunc(new MyStringContainer("Hello"), {
  plus " world!"
})
```

```
String myFunc(MyStringContainer container,
              Closure someClosure){
  cl.delegate = container
  someClosure()  // <- no explicit return
}

//                note paren placement here  |
//                                          V
println myFunc(new MyStringContainer("Hello")) {
  plus " world!"
}    // <- not here
```

In either case, `"Hello world!"` is printed.

Snippet 6: Final closure arguments can be passed after a closing function-call's parentheses, and return values need not be explicitly declared.

It might initially seem like a convoluted markup language, only marginally better than hand-crafting an XML that will be parsed out by our builder. Knowing the concepts presented above, however, we can see that it is a culmination of syntactic sugar and closure delegation. In my own learning of Groovy and Gradle, realizing this did a great deal to demystify the code I was attempting to improve.

```
// From apache/geode.git/geode-assembly/build.gradle
tasks.register('depsJar', Jar) {
  inputs.files {
    configDeps  // <- a task defined elsewhere
  }
  description 'Assembles jar archive defining classpath.'
  archiveName 'geode-dependencies.jar'
  doFirst {
    manifest {
      attributes(
        "Class-Path":
             configDeps.outputs.files.singleFile.text)
    }
  }
}
```

Snippet 7: Via syntactic sugar and closure delegation, a function `register` is invoked with three arguments: a String name, a class, and a closure. Within the context and from the delegate of that closure, the `doFirst` function takes another closure, the delegate of which can handle the `manifest` function call. And so on.

3 Gradle crash course

3.1 The root project, subprojects, and buildSrc

Your Gradle build structures itself similar to your directory structure. The build begins in some root directory containing a `build.gradle` and `settings.gradle` files. This root directory corresponds to your *root project* of the build.

In the `settings.gradle` file, you might indicate subdirectories that this build should `include`. These directories become *subprojects*.

In an ideal world, projects and subprojects are structured hierarchically, with shared resources required by multiple subprojects belonging to their nearest common ancestor. While we rarely get to live in an ideal world, developers should do as much as they can to keep their builds modular and, barring full modularity, looking only upward. When necessary, one subproject can refer to another simply by calling for it by it's colon-separated path, e.g. `project("path:to:some:subproject")`. Be warned, however, that this can easily undermine the stability and correctness of your build, while also making configuration of either subproject more difficult. (Build phases and configuration is discussed more below.)

Lastly, there is one special, optional "side project" – the `buildSrc` project. This project does not need to be called out in your `settings.gradle`, since it isn't intended as part of your project itself. Rather, `buildSrc` will contain any special files that you need to build for the execution of the Gradle build itself. For instance, if you have a custom class defining how testing should work, that class belongs to `buildSrc` subproject. The `buildSrc` project is always built before your main build begins,[5] and so those classes will be available to your root project and subprojects as needed.

[5] Or rather, if your `buildSrc` is relatively stable, it should more often get the required artifacts from the Gradle cache.

3.2 The build in broadest of strokes - Tasks, the Task Graph, and Task I/O

Things get done in Gradle by *tasks*. A task is essentially the atomic unit of work in Gradle.[6] A task typically has some inputs, some outputs, and some task dependencies.

Every task is named. When you invoke a build via `./gradlew build`, you are invoking the `:build` task that lives in the root project. Often, this will require the building of each subproject, and these tasks are also named, with the colon-separated path prepended to the task name. So when you run `./gradlew build`, you will run `:build` and also `:subProject:build` and `:subProject:deeperSubProject:build`

How does the initial `:build` task no to invoke those other tasks? The task graph!

3.3 Phases of a build

The lifecycle of a build[7] consists of three main phases: Initialization, Configuration, and Execution.

The Initialization phase loads your `settings.gradle` to identify what subprojects and (in a multi-project build) what projects will been needed for this build.

4 Advice for Gradle Users

4.1 Gradle gotchas

4.1.1 A task should depend on a task, not on a task.outputs.files

Getting the files output by a task is not the same as getting the outputs directly. For instance,

```
tasks.register('myTask') {
  inputs.files { someOtherTask.outputs.files }
  // task configuration
}
```

does not imply a task dependency in the way that the following does.

```
tasks.register('myTask') {
  inputs.files { someOtherTask }
  // task configuration
}
```

The former declares locations on disk as the input to a task, whereas the latter declares the output of a task as the input to another task. Because Gradle builds incrementally and `inputs` are more concerned with changes than explicit content, the absence of these files is considered a valid input. Depending directly on another task, as in the latter example, had additional safeguards that guarantee that a files are being produced for consumption.

4.2 Gradle best practices and conventions

4.3 Gradle command line interactions

4.4 Important notes for consuming Geode via composite build

[6] This is not precisely accurate. The smallest cachable unit of work in Gradle is an *action*. Most of the time, you won't need to deal with things at such a fine granularity. But if you explicitly declare out a `doFirst` or `doLast` block, you are registering additional actions to your task.

[7] See the docs: `https://docs.gradle.org/current/userguide/build_lifecycle.html`