Scripting with Groovy

A **script** is a tool that automates a sequence of steps[1]. A script may also be called **batch file** or **macro** in some systems. The main idea is to automate tasks that would otherwise have to be performed manually (or interactively with the computer).

Shell scripts

Shell (Bash) scripts are widely used in Bash terminals (available in Linux, MacOS and Windows) and very popular among system administrators. Such UNIX environments provide a wealth of tools that can be used to build your scripts: `grep`, `diff`, `sort`, `make`, `crypt`, `tar`, `sed`, `awk`, `vi`, among others. For example the script below gets the name of the first argument, denoted by `$1`. Then it creates an empty file with that file name, adding the extension `.txt`, under the folder `./temp`.

```
#!/bin/bash
filename=$1
mkdir ./temp
touch ./temp/$filename.txt
```

The first line `#!/bin/bash` (usually called shebang) locates which interpreter to use for executing the script.

Scripting languages

Shell scripts are useful when the target environment provides the toolkit required and when the sequence of steps to be performed are short. For complex scripts, scripting languages provide better mechanisms for constructing the script (modularity, object-oriented and functional features, libraries, among others). A **scripting language** is a high-level programming language that is usually interpreted, instead of being compiled. In addition, scripting languages are geared towards automation of data manipulation tasks, such as accessing files, processing text with regular expressions and cleansing data.

We are going to use Groovy 2.4.15, which is

- is an agile and dynamic language for the Java Virtual Machine;
- builds upon the strengths of Java but has additional power features inspired by languages like Python, Ruby and Smalltalk;
- makes modern programming features available to Java developers with almost-zero learning curve;
- provides the ability to statically type check and statically compile your code for robustness and performance;

- supports compact syntax so your code becomes easy to read and maintain;
- makes writing shell and build scripts easy with its powerful processing primitives, OO abilities and an Ant DSL;
- increases developer productivity by reducing scaffolding code when developing web, GUI, database or console applications;
- simplifies testing by supporting unit testing and mocking out-of-the-box;
- seamlessly integrates with all existing Java classes and libraries;
- compiles straight to Java bytecode so you can use it anywhere you can use Java.

That is, Groovy provides succinct syntax to use JVM libraries - like the Java Collections Framework - that you already know from Java. In addition, Groovy will be used to program Gradle build scripts and is used to test Java applications with the Spock Framework, which will be used in sprint three. The features of the Groovy language that you are going to learn are also found, to some extent, in other scripting languages, such as Python and Ruby.

The following Groovy script is equivalent to the bash script above:

```
#!/usr/bin/env groovy
def filename = new File(args[0]).name
new File("./temp").mkdir()
new File("./temp/${filename}.txt").createNewFile()
```

In this script, instead of using bash tools, we use Java libraries to perform tasks. Depending on the task to be performed, this may be an advantage, if an appropriate tool does not exist in the bash environment and this is available as a Java library (or more generically, a JVM library).

Groovy can be used as a server-side scripting language, that is to program the logic behind a web server, and there are mature Groovy frameworks used with this purpose. For example, Grails is used to develop web applications based on Spring Boot, the Java framework that we are going to use. For pedagogic reasons, we are, however, going to keep this criteria in mind:

- Java will be used to program web application logic, using Spring Boot. This will allow us to make connections with other modules, where concurrency is discussed. Besides we will be able to use code analysis tools, like code coverage, which are available for Java programs but not for scripting languages.
- Groovy will be used in scripting tasks: for writing Gradle build scripts and for testing.

Groovy

In this section, we provide a summary of operators with examples, extracted from the DZone Refcardz on Groovy by Dierk König. Each type of operator

is linked to the corresponding section in the Groovy documentation where the operator is explained, signposted with the icon :books:.

:loudspeaker: **There is no need to memorise these operators but you should be able both to use them and to spot when a operator is not used correctly.**

Language

The following concepts should be easy to recall from your background in object-oriented programming:

- :books: doc on classes (we will only need normal and abstract classes)
- :books: doc on fields
- :books: doc on methods

Strings

In Groovy, we can use Java Strings with single quotes `'..'` and with triple single quotes `'''...'''` for multiline strings (or text). Strings are literals. :books: doc on strings

We can also use GStrings, which are used for writing templates. GStrings are written with double quotes `".."` and with triple double quotes `"""..."""` for multiline GStrings (or text). Placeholders are declared using variables enclosed in between `${..}`. When a GString is evaluated, Groovy evaluates the expressions `${..}.toString()` and **interpolates** them in the template. That is, templates are instantiated by replacing expressions `${..}` with their resulting evaluated value. :books: doc on GStrings

Overloaded operators

Some of the most common arithmetic and relational operators:

Operator	Method
a + b	a.plus(b)
a – b	a.minus(b)
a * b	a.multiply(b)
a / b	a.div(b)
a % b	a.mod(b)
a++, ++a	a.next()
a–, –a	a.previous()
a**b	a.power(b)
a | b	a.or(b)

Operator	Method
a & b	a.and(b)
a ^ b	a.xor(b)
a == b	a.equals(b)
a != b	! a.equals(b)
a <=> b	a.compareTo(b)
a > b	a.compareTo(b) > 0
a >= b	a.compareTo(b) >= 0
a < b	a.compareTo(b) < 0
a <= b	a.compareTo(b) <= 0
a as B	a.asType(B)

Documentation:

- :books: doc on arithmetic operators
- :books: doc on relational operators
- :books: doc on logical operators

Special operators

Operator	Method	Name
a ? b : c	if (a) b else c	ternary if
a ?: b	a!=null ? a : b	Elvis
a?.b	a==null ? a : a.b	null safe
switch(a){ case b: ... } *or* if(a in b)	b.isCase(a) // b is a class	switch case by type
a as B	a.asType(B)	casting

Documentation:

- :books: doc on control structures
- :books: doc on object operators

Collections

Ranges

Ranges appear inclusively like `0..10` or half-exclusively like `0..<10`. They are often enclosed in parentheses since the range operator has low precedence.

```
assert (0..10).contains(5)
assert (0.0..10.0).containsWithinBounds(3.5)
for (item in 0..10) {
    println item
```

```
}
for (item in 10..0) {
    println item
}
(0..<10).each { println it }
```

Integer ranges are often used for selecting *sublists*. Range boundaries can be of any type that defines `previous()`, `next()` and implements `Comparable`. Notable examples are String and Date.

:books: examples with ranges

Lists

Lists look like arrays but are of type `java.util.List` plus new enhanced methods.

```
[1,2,3,4] == [1..4]
[1,2,3] + [1] == [1,2,3,1]
[1,2,3] << 1 == [1,2,3,1]
[1,2,3,1] - [1] == [2,3]
[1,2,3] * 2 == [1,2,3,1,2,3]
[1,[2,3]].flatten() == [1,2,3]
[1,2,3].reverse() == [3,2,1]
[1,2,3].disjoint([4,5,6]) == true
[1,2,3].intersect([4,3,1]) == [3,1]
[1,2,3].collect{ it+3 } == [4,5,6]
[1,2,3,1].unique().size() == 3
[1,2,3,1].count(1) == 2
[1,2,3,4].min() == 1
[1,2,3,4].max() == 4
[1,2,3,4].sum() == 10
[4,2,1,3].sort() == [1,2,3,4]
[4,2,1,3].findAll{it%2 == 0} == [4,2]
def anims=['cat','kangaroo','koala']
anims[2] == 'koala'
def kanims = anims[1..2]
anims.findAll{ it.startsWith('k') } == kanims
anims.find{ it.startsWith('k') } == kanims[0]
```

Documentation on lists:

- :books: doc on lists
- :books: examples with lists

Maps

Maps are like lists that have an arbitrary type of key instead of integer. Therefore, the syntax is very much aligned.

```
def map = [a:0, b:1]
```

Maps can be accessed in a conventional square-bracket syntax or as if the key was a property of the map.

```
assert map['a'] == 0 // a[b] is a.getAt(b)
assert map.b == 1
map['a'] = 'x' // a[b] = c  is a.putAt(b, c)
map.b = 'y'
assert map == [a:'x', b:'y']
```

There is also an explicit get method that optionally takes a default value.

```
assert map.c == null
assert map.get('c',2) == 2
assert map.c == 2
```

Map iteration methods take the nature of `Map.Entry` objects into account.

```
map.each { entry ->
    println entry.key
    println entry.value
}
map.each { key, value ->
    println "$key $value"
}
for (entry in map) {
    println "$entry.key $entry.value"
}
```

Documentation:

- :books: doc on maps
- :books: examples with maps

Closures

Closures capture a piece of logic and the enclosing scope. They are first-class objects and can receive messages, can be returned from method calls, stored in fields, and used as arguments to a method call.

Use in method parameter

```
def forEach(int i, Closure yield){
    for (x in 1..i) yield(x)
}
```

Use as last method argument

```
forEach(3) { num -> println num }
```

Construct and assign to local variable

```
def squareIt = { println it * it} forEach(3, squareIt)
```

Bind leftmost closure param to fixed argument

```
def multIt = {x, y -> println x * y}
forEach 3, multIt.curry(2)
forEach 3, multIt.curry('-')
```

Closure parameter list examples:

Table 3: books: doc on closures

Closure	Parameters
{ ... }	zero or one (implicit 'it')
{-> ... }	zero
{x -> ... }	one
{x=1 -> ... }	one or zero with default
{x,y -> ... }	two
{ String x -> ... }	one with static type

Files

Often used reading methods

```
def file = new File('/data.txt')
println file.text // for type File, Reader, URL, InputStream,Process
def listOfLines = file.readLines()
file.eachLine { line -> ... }
```

Often-used writing methods

```
out << 'content' // for type File, Writer, OutputStream, Socket, and Process
```

Reading and writing with Strings

```
def out = new StringWriter()
out << 'something'
def str = out.toString()
def rdr = new StringReader(str)
println rdr.readLines()
```

:books: doc on working with files

References

[1] S. McConnell. *Code Complete, Second Edition.* Microsoft Press, Redmond, WA, USA, 2004.