Groovy for system scripting

Greach 2017 Lightning talk

This talk is focused on learning how to write Groovy scripts. I'll show how to write Groovy scripts which interact with the operating system among other tasks. Everything will be guided by an example to show the points of interest to keep in mind when creating a script step-by-step.

Intro

- Tomás Crespo García
- Software developer at 

- Talk available at 

https://github.com/tcrespog/greach-2017-groovy-scripting

Motivation

- Typical features in scripting languages:


Task automation


System control


Distributed as source code

A scripting language tipically has the following three fundamental features:

1. It allows task automation, in other words, it allows to solve problems without user interaction.
2. It allows to control a specific system, for example, JavaScript is a scripting language that allows to control the web browser, other scripting languages like Lua allow to control larger software applications, finally, some scripting languages allow to control the operating system.
3. Software written in scripting languages is distributed as source code rather than as a binary file, ready to be edited and executed. No compilation phase involved, at least to be performed explicitly by the user.

Motivation

- Popular scripting languages for operating sytems:


Python


Bash

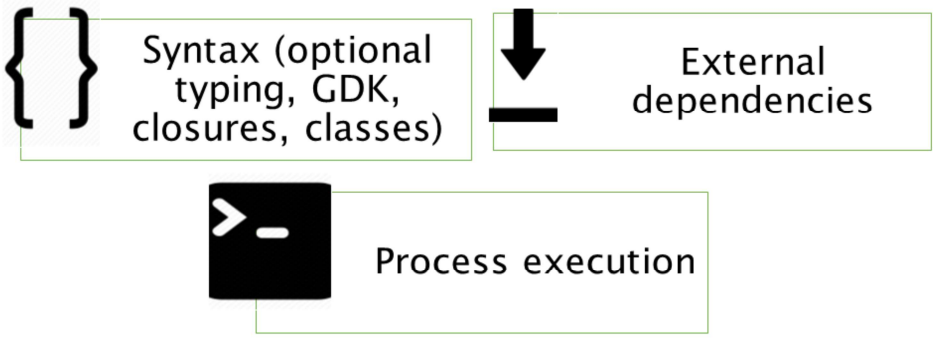

PowerShell

Among all the scripting languages for operating systems, we can find the following popular ones:

1. Python is a popular scripting language executed via an interpreter which usually comes preinstalled in some Linux distributions.
2. Bash is the language used in the shell interpreter, other shell and their languages like C-Shell or zsh may be used in other Unix-like environments.
3. PowerShell is the language for the Windows operating system's modern shell interpreter.

Motivation

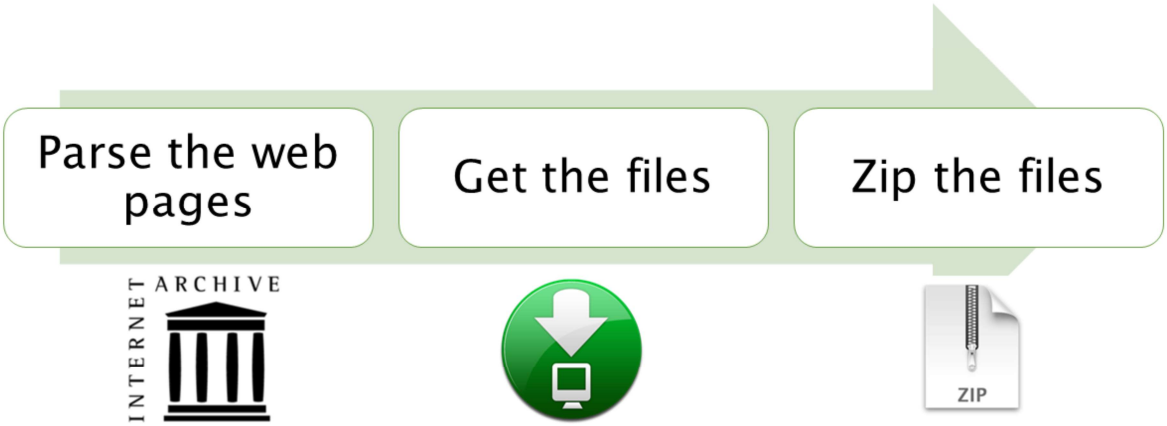
- Use  as a scripting language.
- Advantages:



Groovy can also be used as a scripting language for operating systems, it fullfils the three features of scripting languages mentioned earlier.
With Groovy we can take advantage of its concise, readable and expressive syntax; download external dependencies/libraries to perform tasks; and execute external processes (just like in other scripting languages).

Guided example

- File crawler and bundler on a Linux system.



The guided example which I'll use to explain points of interest will consist of a script which will crawl and bundle some files from the Internet Archive, a digital library that gathers free public access documents.
In order to do this:

1. We parse some web pages to extract the link adresses which point to the files.
2. We download those files.
3. We zip all the downloaded files

Preparing the environment

- Required software:

Groovy

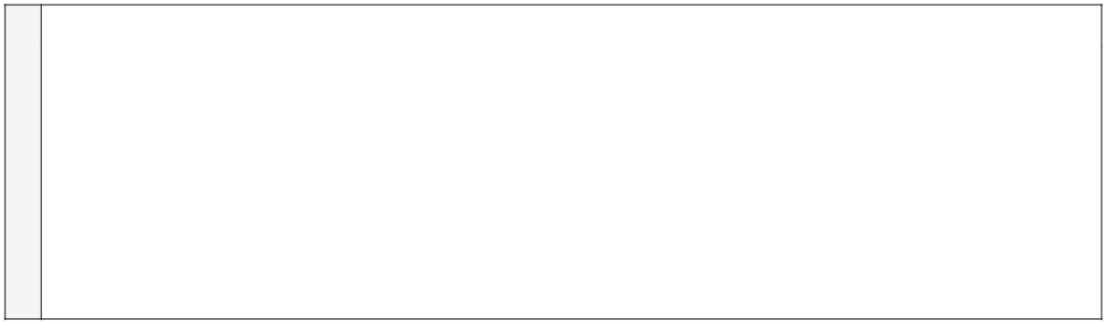
JDK

Of course in order to run Groovy scripts we need the Groovy binary and the Java Development Kit installed in our system.
Which version? Each version of Groovy recommends a version of the JDK. As of today, the current version of Groovy (2.4.10) recommends Java 8.

Creating the Groovy script

- **Step 1**: create an empty file

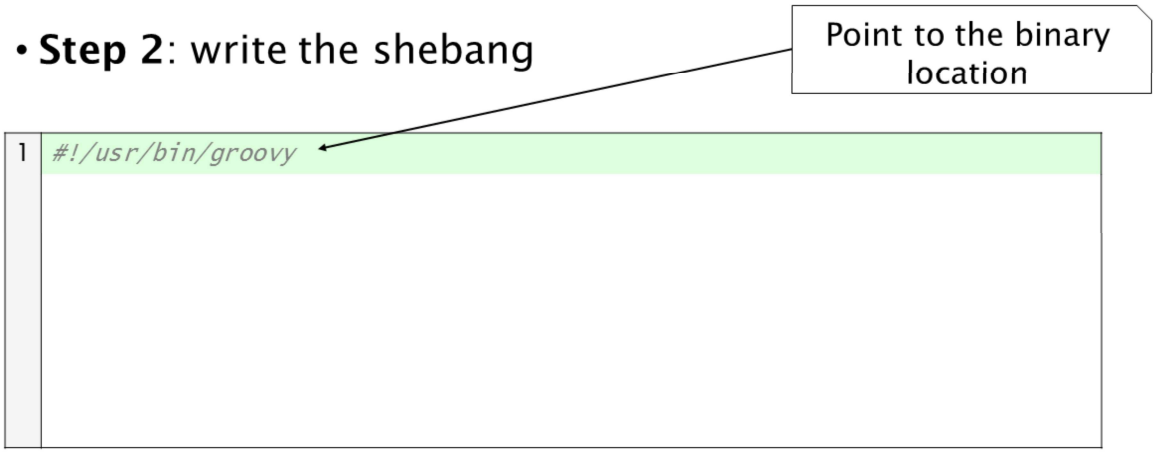
script.groovy



The first step to write a Groovy script is to create and empty file. By convention, the extension to write Groovy scripts is .groovy.

Coding the Groovy script

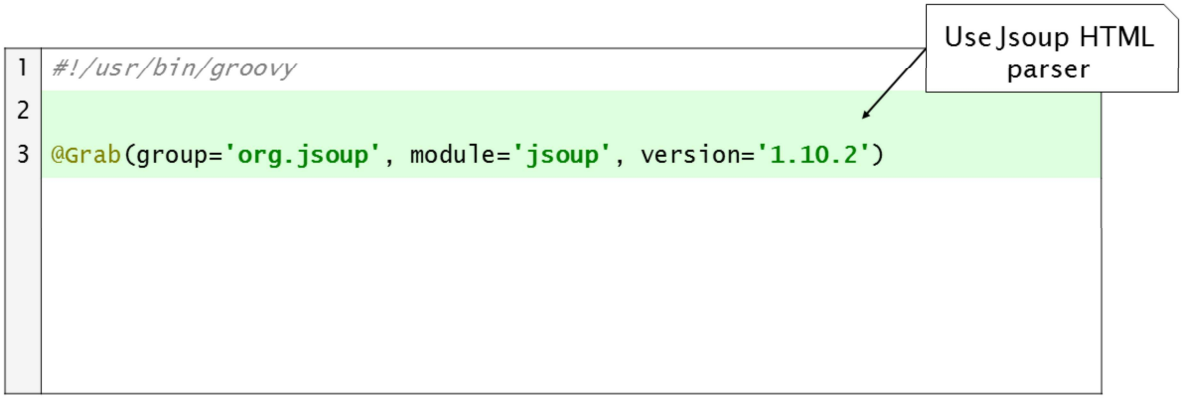
- **Step 2**: write the shebang



The shebang character sequence (hash symbol and exclamation mark symbol) allows the system to recognize the program used to execute the script in Unix-like operating systems.

Coding the Groovy script

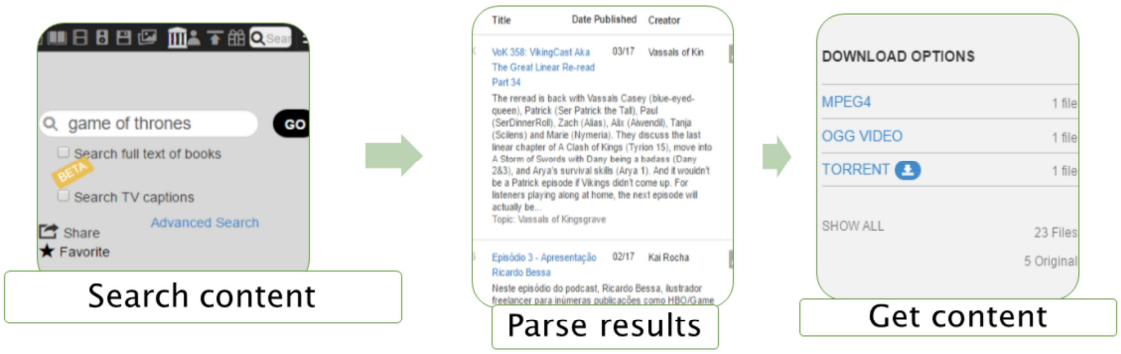
- **Step 3**: use external dependencies with Grapes



Grapes is a dependency manager embedded into Groovy. It provides annotations that let you download Maven dependencies to be used in the script.
With the Grab annotion you just have to specifiy the “Maven coordinates” of the library you want to use and it will be downloaded or resolved locally when the script is executed.
The default repository is the Maven central repository, but we can also specify a custom Maven repository using other annotations.

Coding the Groovy script

- **Step 4**: write custom logic.



So, remember what we want to do as part of the example script. We want to search in Archive.org about a specific content. Parse the HTML results to extract the URLs to each file and then download the files.
It makes sense to write a Crawler class to do the job.

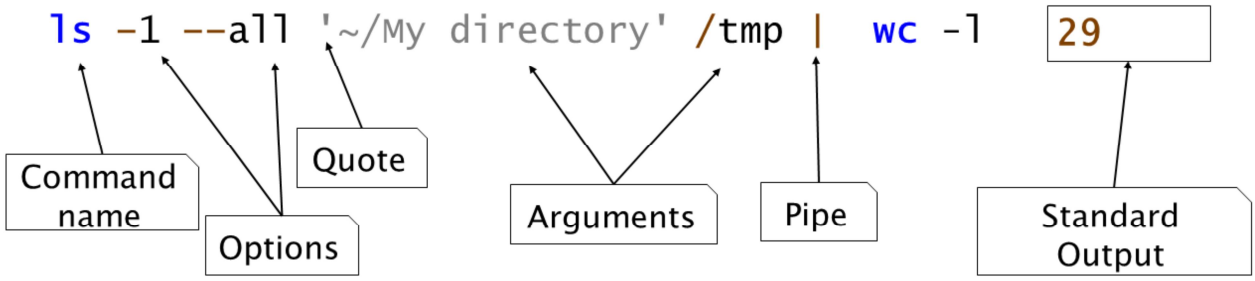
We'll download the files in torrent format, because they'll be smaller in size, therefore they'll take less time to download.

```
 5  import org.jsoup.Document
    ...
 9  class Crawler {
10
11      final static String BASE_URL = 'https://archive.org'
12      Document resultsListPage
13      List<String> resultsTorrentFilesUrls
14
15      List<String> searchAndDownloadTorrentFiles(String text) {
16          downloadResultsPage(text)
17          gatherResultsTorrentFilesUrls()
18          downloadTorrentFiles()
19      }
    ...
70  }
```

This code shows the process drawn on the previous slide, some details are ommited, but represents the way you can write classes and methods in the script to solve the problem by separating responsibilities using object-oriented principles.

Coding the Groovy script

- **Step 5**: execute external processes.
- Command syntax reminder:



How do we execute external processes using Groovy?
Well, when we execute a command in a shell we write lines with this appearance.
Each command line could have the following parts:

1. The command name is the first part of the command and represents the program to be searched and executed.
2. Options modify how the command should do its job.
3. Arguments are used to pass information to the command.
4. Quotes help to ignore the special meaning of some characters (metacharacters).
5. The pipe symbol is used in shells to redirect the output of a command as the input of another command.
6. The executed program can print information to standard output or standard error depending on the outcome of the process.

The shell interprets the command line, finds the program and sends a copy of the options and arguments, it's up to the program to react to them properly.

Coding the Groovy script

- Execute processes with Groovy (GDK convenient methods):

```
Process process = "ls -la 'My directory'".execute()
//Process process = ['ls', '-la', 'My directory'].execute()
println process.text
```

Run and get standard output

java.lang.String	execute				
java.util.List	execute				
java.lang.Process	getText	waitForProcessOutput	pipeTo	waitForOrKill	…

In order to execute external processes from Groovy we can write a command line in a java.lang.String or a java.util.List implementation and get an abstract handler to the process by calling the execute() method. This java.lang.Process handler will allow us to control the external process execution (get the output, pipe standard output to another process, get its return value, …) through the convenient methods provided by GDK.

Writing a command line in a List allows us to don't worry about quoting in order to scape blank characters

```
5   import org.jsoup.Document
    ...
9   class Crawler {
    ...
69      void downloadTorrentFiles() {
70          List<String> command = ['wget', '-nc', '-nv'] + resultsTorrentFilesUrls
71          Process process = command.execute()
72
73          StringBuffer error = new StringBuffer()
74          process.waitForProcessOutput(null, error)
75
76          println error.toString()
77      }
78
79  }
```

Run and get standard error output

We use the wget utility in order to download all the collected torrent files.

First, we build the command we want to execute, then we run it and we get the standard error output. The wget utility prints its output to standard error, so we are ignoring standard output.

Coding the Groovy script

- **Step 6**: piping processes (`pipeTo` method, | operator).

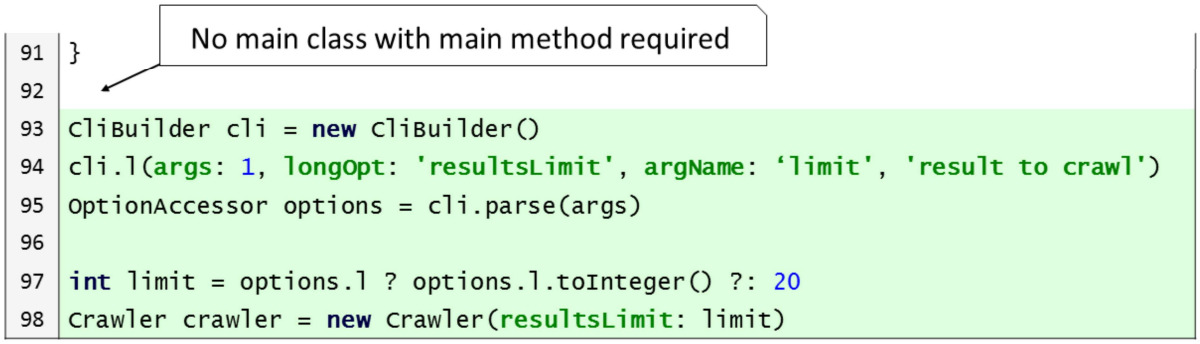
```
81  void zipFiles() {
82      Process listFilesProcess = 'ls'.execute()
83      Process filterTorrentsProcess = 'grep .torrent'.execute()
84
85      listFilesProcess | filterTorrentsProcess
86
87      List<String> torrentFiles = filterTorrentsProcess.text.split('\n')
88
89      Process zipProcess = (['zip', 'bundle.zip'] + torrentFiles).execute()
90      println zipProcess.text
91  }
```

The GDK provides the pipeTo method (or pipe operator) in the java.langProcess API, which allows to redirect the output of a process as the input of another process.

As you can see in the sample code, in order to find the files we want to zip, we chain the ls program output as the input of the grep fiilter, then we run and get the standard output of the last process in the chain or pipeline.

Coding the Groovy script

- **Step 7**: reading command line arguments.
- Process command line arguments with `groovy.util.CliBuilder`



Sometimes is useful to pass arguments to a script. In order to do this we can make use of the groovy.util.CliBuilder class with let us process options and arguments in a expressive way. It also allows you to write the help information for the script in a quick way.

It isn't required to write a main class with a main method in the script, however, the args array, which contains all the options and arguments sent to the script, is available implicitly.

Executing the Groovy script

- `$ groovy script.groovy -l 5`
- `$ ./script.groovy -l 5`

Run the script from your shell directly or via the Groovy binary passing the Groovy script file as an argument. Make sure the script has the execution permission properly set for the current user.

Left in the inkwell

- Read environment variables.

```
String pathValue = System.getenv('PATH')
```

- Print to standard error.

```
System.err.println('Print to standard error')
```

- Read standard input.

```
BufferedReader inputReader = new BufferedReader(new InputStreamReader(System.in))
String input = inputReader.text
```

These are some points of interest that were left unsaid (left in the inkwell metaphorically) from the guided example, but are useful to use in scripts.

1. We can read environment variables using System.getenv
2. We can print to standard error printing to Sytem.err.
3. We can read from standard input using the reader classes over System.in.

Left in the inkwell

- Execute processes in Windows systems.

```
Process echoCmd = ['cmd', '/c', 'echo', 'Hello Windows'].execute()
println echoCmd.text

Process echoPowershell = ['powershell', 'echo', 'Hello Windows'].execute()
println echoPowershell.text
```

Specify the command interpreter explicitly

In order to execute processes in Windows we have to specify the command interpreter explicitly at the beginning of the command line.

Conclusions

- Groovy serves **nicely** as a scripting language.
- Try **not** to write **huge scripts**.
- Consider **mixing** Groovy with other scripting languages.
- **Learn more** at the GDK API:
  - `java.util.Process`
  - `groovy.lang.Grapes`
  - `groovy.util.CliBuilder`

Finally, some conclusions we extract from using Groovy as a scripting language:

1. Groovy lets you perform the same core tasks as other scripting languages.
2. When a script is getting big, consider building a full project with build management tool support or write several scripts with different responsibilities.
3. Write scripts in different languages and leverage their strengths.
4. Explore and dive deeper in Groovy scripting.

It's over!
Thank you!

https://github.com/tcrespog/greach-2017-groovy-scripting

Greach 2017 

Lightning talk