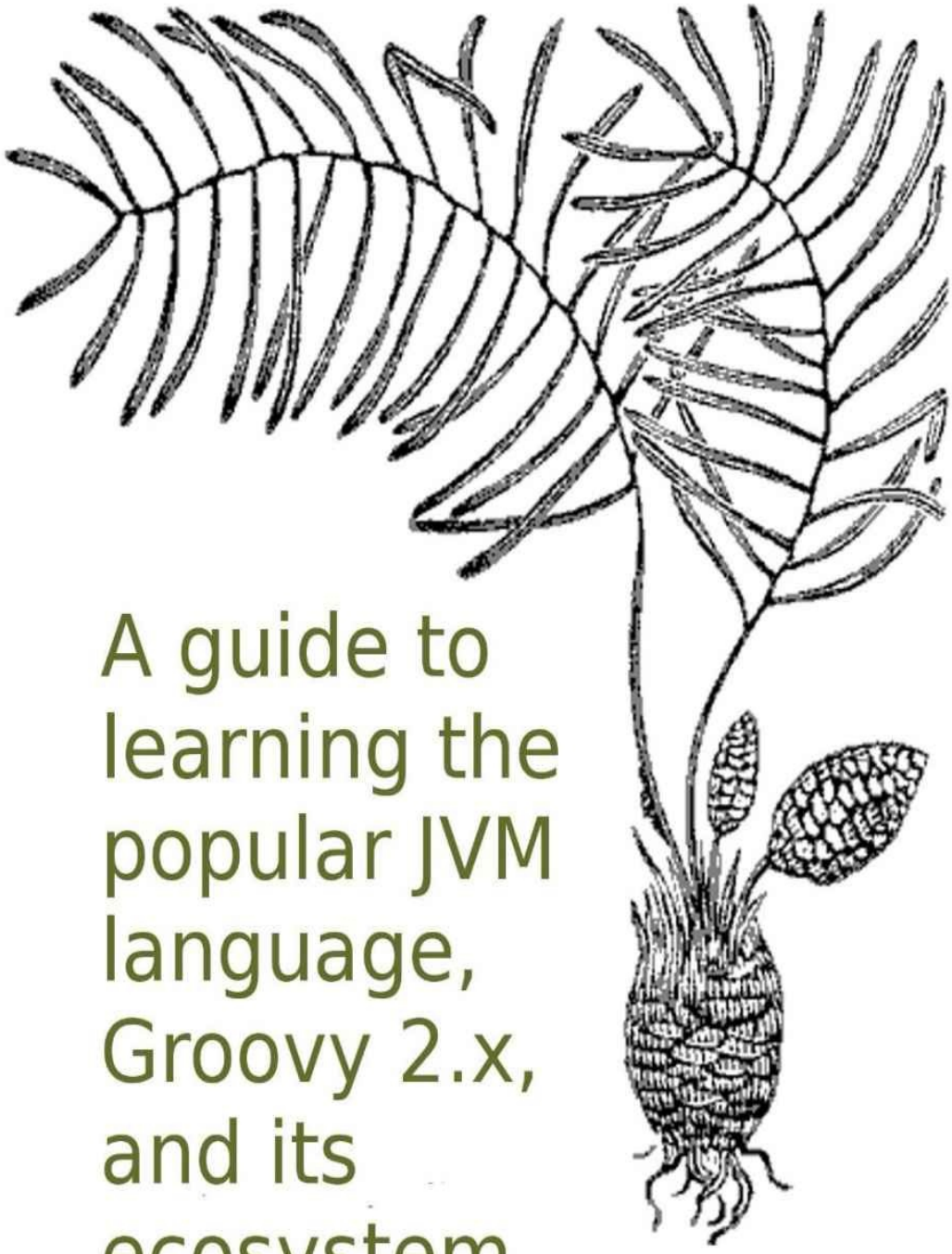
Learning Groovy



A guide to learning the popular JVM language, Groovy 2.x, and its ecosystem

Adam L. Davis

Learning Groovy

A guide to learning the popular JVM programming language, Groovy 2.x, and its ecosystem

Adam L. Davis

This book is for sale at http://leanpub.com/learninggroovy

This version was published on 2016-03-02

* * * * *

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

* * * * *

To my parents, to whom I owe everything in ways large and small.

To my children, who mean the world to me.

Table of Contents

Getting Groovy

About This Book

This book is organized into several chapters, starting from the most basic concepts. If you already understand a concept, you can safely move ahead to the next chapter. Although this book concentrates on Groovy, it will also refer to other languages such as Java, Scala, and JavaScript.

As the title suggests, this book is about learning Groovy, but will also cover related technology such as build tools and web frameworks.

Assumptions

This book assumes the reader already is familiar with Java's syntax and basic programming ideas.

Icons

Tips

If you see text stylized like this, it is extra information that you might find helpful.

Info

Text stylized this way is usually a reference to additional information for the curious reader.

Warnings

Texts like this are cautions for the wary reader - many have fallen on the path of computer programming.

Exercises

This is an exercise. We learn by doing. These are highly recommended.

Software to Install

Before you begin programming, you need to install some basic tools.

Java/Groovy

For Java and Groovy you will need to install the following:

- JDK (Java Development Kit) such as JDK 8.
- IDE (Integrated Development Environment) such as NetBeans 8.
- Groovy

Install Java and NetBeans 8

Download and install the Java JDK 8 with NetBeans. Open NetBeans and select “File” ⇒ “New Project…” ⇒ “Java Application”

Install Groovy

Go and install Groovy.

You can Download Groovy and install it or you could use SDKMAN.

You may need to set the `JAVA_HOME` environment variable to the location of your Java installation.

Trying it out

After installing Groovy, you should use it to try coding. Open a command prompt and type `groovyConsole` and hit enter to begin.

In the groovyConsole, type the following and then hit `Ctrl+r` to run the code:

```
1 print "hello"
```

You should keep the “Groovy Console” open and use it to try out all of the examples from this book.

Others

Once you have the above installed, you should probably install:

- SDKMAN: The Software Development Kit Manager.
- Git: A version control program.

Go ahead and install these if you're in the mood - I'll wait.

Code on Github

A lot of the code from this book is available on github.com/adamldavis/learning-groovy. You can go there at any time to follow along with the book.

Groovy 101

What is Groovy?

Groovy is a flexible open-source language built for the JVM (Java Virtual Machine) with a Java-like syntax. It can be used dynamically or statically-typed, your choice. It supports functional programming constructs, including first-class functions, currying, and more. It has multiple-inheritance, type inference, and meta-programming.

Groovy began in 2003 partly as a response to Ruby. Its main features were dynamic typing, meta-programming (the ability to change classes at runtime) and tight integration with Java. Although its original developer abandoned it, many developers in the community have contributed to it over the years. Various organizations have supported the development of Groovy in the past and like many open-source projects, it can not be attributed to one person or company. It is now an Apache Software Foundation project.

Groovy is very similar in syntax to Java so it is generally easy for Java developers to learn (Java code is generally valid Groovy code). However, Groovy has many additional features and relaxed syntax rules: closures, dynamic typing, meta-programming (via metaClass), semicolons are optional, regex-support, operator overloading, GStrings, and more. Groovy is interpreted at runtime, but in Groovy 2.0 the ability to compile to byte-code and enforce type-checking were added to the language.

Compact Syntax

Groovy's syntax can be made far more compact than Java. For example, the following code in Standard Java 5+:

```
1 for (String it : new String[] {"Rod", "Carlos", "Chris"})
2       if (it.length() <= 4)
3               System.out.println(it);
```

can be expressed in Groovy in one line as:

```
1 ["Rod", "Carlos", "Chris"].findAll{it.size() <= 4}.each{println it}
```

It has tons of built-in features to make the above possible (compact List definition, extensions to JDK objects, closures, optional semi-colons, and the println method, optional parenthesis).

Dynamic Def

A key feature of Groovy is available dynamic typing using the `def` keyword. This keyword replaces any type definition allowing variables to be of any type. This is somewhat like defining variables as Object but not exactly the same because the Groovy compiler treats `def` differently. For example you can use `def` and still use the `@TypeChecked` annotation, which we will cover later.

List and Map Definitions

Groovy makes List and Map definitions much more concise and simple. You simply use brackets ([]) and the mapping (:) symbol for mapping keys to values:

```
1 def list = [1, 2]
2 def map = [cars: 2, boats: 3]
3 println list.getClass() // java.util.ArrayList
4 println map.getClass() // java.util.LinkedHashMap
```

By default, Groovy interprets Map key values as strings without requiring quotes. When working with maps with String keys, Groovy makes life much easier by allowing you to refer to keys using dot-notation (avoiding the get and put methods). For example:

```
1 map.cars = 2
2 map.boats = 3
3 map.planes = 0
4 println map.cars // 2
```

This even makes it possible to Mock objects using a Map when testing.

Groovy lists even overrides the “left shift” operator (<<) which allows the following syntax example:

```
1 def list = []
2 list.add(new Vampire("Count Dracula", 1897))
3 // or
4 list << new Vampire("Count Dracula", 1897)
5 // or
6 list += new Vampire("Count Dracula", 1897)
```

Groovy allows overriding of common operators like plus and minus. We will cover this in a later section.

Groovy GDK

Built-in Groovy types (the GDK) are much the same as Java’s except that Groovy adds tons of methods to every class.

For example, the `each` method is added which allows you to iterate over a collection as follows:

```
1 ["Java", "Groovy", "Scala"].each{ println it }
```

The `println` and `print` methods are short-hand for calling those methods on `System.out`.

We will cover the GDK more in depth later.

Everything is an Object

Unlike in Java, primitives can be used like Objects at any time so there appears to be no distinction.

For example, since the GDK adds the `times` method to Number you can do the following:

```
1 100.times { println "hi" }
```

This would print “hi” one hundred times.

Easy Properties

Groovy takes the idea of Java Beans to a whole new level. You can get and set bean properties using dot-notation (and Groovy automatically adds getters and setters to your classes if you don’t).

For example, instead of `person.getFirstName()` you can use `person.firstName`. When setting properties, instead of `person.setFirstName("Bob")` you can just use `person.firstName = 'Bob'`.

Public Parts

Unlike Java, Groovy always defaults to public.

You can also easily get a list of all properties of an object in Groovy using `.properties`. For example:

```
1 println person.properties
```

Use `properties` to explore some class in Groovy that you want to know more about.

GString

Groovy adds its own class, called `GString`, that allows you to embed groovy code within strings. This is another features that makes Groovy very concise and easy to read.

For example, it makes it easy to embed a bunch of variables into a string:

```
1 def os = 'Linux'
2 def cores = 2
3 println("Cores: $cores, OS: $os, Time: ${new Date()}")
```

The dollar $ allows you to refer directly to variables, and ${code} allows you to execute arbitrary Groovy code.

String

If you just want to use a `java.lang.String`, you should use single quotes (‘foo’).

Closures

A closure is a block of code in Groovy, which may or may not take parameters and return a value. It’s similar to lambda expressions in Java 8 or an inner class with one method.

Groovy closures have several implicit variables:

- `it` - If the closure has one argument, it can be referred to implicitly as **it**
- `this` - Refers to the enclosing class.
- `owner` - The same as `this` unless it is enclosed in another closure.
- `delegate` - Usually the same as `owner` but you can change it (this allows the methods of *delegate* to be in scope).

Closures can be passed as method arguments. When this is done (*and it is the last argument*), it may go outside the parentheses. For example:

```
1 def list = ['foo','bar']
2 def newList = []
3 list.collect( newList ) {
4   it.toUpperCase()
5 }
6 println newList //  ["FOO", "BAR"]
```

Return Optional

The `return` keyword is completely optional in Groovy. A method or closure simply returns its last expression.

You can also assign closures to variables and call them later:

```
1 def closr = {x -> x + 1}
2 println( closr(2) )
```

Closure Exercise

Create a Closure and print out its class using `getClass()`.

A Better Switch

Groovy's switch statement is much like Java's, except that it allows many more `case` expressions. For example, it allows Strings, lists, ranges, and class types:

```
 1 switch ( x ) {
 2 case "foo":
 3     result = "found foo"
 4      break
 5 case [4, 5, 6]:
 6     result = "4 5 or 6"
 7     break
 8 case 12..30: // Range
 9     result = "12 to 30"
10     break
11 case Integer:
12     result = "was integer"
13     break
14 case Number:
15     result = "was number"
16     break
17 default:
18     result = "default"
19 }
```

Meta-programming

In Groovy you can add methods to classes at runtime - even to core Java libraries. For example, the following code adds the `upper` method to the String class:

```
1 String.metaClass.upper = { -> toUpperCase() }
```

or for a single instance (`str`):

```
1 str.metaClass.upper = { -> toUpperCase() }
```

The `upper` method would convert the String to upper-case:

```
1 str.upper() == str.toUpperCase()
```

Static Type Checking

If you add the `@TypeChecked` annotation to your class, it causes the compiler to enforce compile time type-checking. It will infer types for you, so your code can still be *Groovy*. It infers the Lowest Upper Bound (LUB) type based on your code.

Gotcha's

- Runtime meta-programming won't work!
- Explicit type needed in closure: `a.collect {String it -> it.toUpperCase()}`

If you add the `@StaticCompile` annotation to your class or method it causes the compiler to compile your Groovy code to byte-code. The generated byte-code is almost identical to compiled Java. Both annotations are located in the `groovy.transform` package. For example:

```
1 import groovy.transform.*
2 @CompileStatic
3 class Foo {
4     List<Integer> nums = [1, 2, 3]
5 }
```

Try out using `@TypeChecked` and using the `def` keyword. It works surprisingly well.

Elvis Operator

The elvis operator was born from a common idiom in Java: using the ternary operation to provide a default value, for example:

```
1 String name = person.getName() == null ? "Bob" : person.getName();
```

Instead in Groovy you can just use the elvis operator:

```
1 String name = person.getName() ?: "Bob"
```

Safe Dereference Operator

Similar to the elvis operator, Groovy has the safe dereference operator which allows you to easily avoid null-pointer exceptions. It consists of simply adding a question mark. For example:

```
1 String name = person?.getName()
```

This would simply set name to null if person is null. It also works for method calls.

Write some Groovy code using Elvis operators and safe dereference several times until you memorize the syntax.

A Brief History

What follows is a brief history of updates to the Groovy language starting with `Groovy 1.8`. This will help you if you must use an older version of Groovy or if you haven't looked at Groovy in several years.

Groovy 1.8

- Command Chains: `pull request on github => pull(request).on(github)`
- GPars is bundled for parallel and concurrent paradigms.
- Closure annotation parameters: `@Invariant({number >= 0})`
- Closure memoization: `{...}.memoize()`
- Built-in JSON support - consuming, producing, pretty-printing
- New AST Transformations: `@Log, @Field, @AutoClone, @AutoExternalizable, ...`

Groovy 1.8 further improved the ability to omit unnecessary punctuation by supporting "Command Chains" which allows you to omit parentheses and dots for a chain of method calls.

Groovy 2.0

- More modular - multiple jars - create your own module (Extension modules)
- `@StaticCompile` - compiles your Groovy code to byte-code.
- `@TypeChecked` - enforces compile time type-checking.
- Java 7 alignments - project Coin & invoke dynamic.
- Java 7 - `catch (Exception1 | Exception2 e) {}`

The huge change in Groovy 2 was the addition of the @StaticCompile and @TypeChecked annotations, which were already covered.

Groovy 2.1

- Full support for the JDK 7 “invoke dynamic” instruction and API.
- Groovy 2.1’s distribution bundles the concurrency library GPars 1.0
- `@DelegatesTo` - a special annotation for closure delegate based DSLs and static type checker extensions.
- Compile-time Meta-annotations - `@groovy.transform.AnnotationCollector` can be used to create a meta-annotation.
- Many improvements to existing AST transformations.

This release saw a huge improvement in performance by taking advantage of Java 7’s invoke dynamic. However, it is not enabled by default (this will be covered later; basically you just have to “turn it on”).

Groovy 2.2

- Implicit closure coercion
- @Memoized AST transformation for methods
- Bintray’s JCenter repository
- Define base script classes with an annotation
- New DelegatingScript base class for scripts
- New @Log variant for the Log4j2 logging framework
- @DelegatesTo with generics type tokens
- Precompiled type checking extensions

The main thing to notice here is implicit closure coercion which allows you to use closures anywhere a SAM (single abstract method interface) could be used. Before this release you would need to cast the closure explicitly.

Groovy 2.3

- Official support for running Groovy on JDK 8
- Traits
- New and updated AST transformations

This release added a brand new concept (Traits) and the “trait” keyword. We will cover them in a later chapter.

Groovy 2.4

- Android support
- Performance improvements and reduced bytecode
- Traits @SelfType annotation
- GDK improvements
- More AST transformations

Out-doing themselves yet again, the developers behind Groovy made tons of improvements and Android support.

Summary

In this chapter you've learned about:

- How Groovy extends and simplifies programming on the JDK in a familiar syntax.
- Groovy's dynamic `def`, easy properties, Closures, a better "switch", meta-programming, type-checking and static-compile annotations, Elvis operator, and safe dereference.
- The concept of "Groovy Truth"
- A brief overview of the features available to Groovy and in what versions they were introduced.

Tools

In addition to `groovy` the Groovy installation comes with several helpful tools.

Console

```
1 groovyConsole
```

The *Groovy Console* is a quick and easy way to try out things in Groovy visually without the over-head of a complete IDE.

Whenever you have an idea you want to try out quickly, open up the Groovy Console, type some code, and then hit `Ctrl-R` to run it. After reading your output and changing the code, hit `Ctrl-W` to clear the output and `Ctrl-R` again to run the code. Once you get used to those two shortcuts the Groovy Console might become an indispensable development tool.

It also has the ability (among other things) to inspect the AST (Abstract Syntax Tree) of your code. Use `Script - Inspect Ast` or `Ctrl-T` to open the Groovy AST Browser.

You can provide a classpath to be available at runtime to the Groovy Console using the `-cp` option. For example, in a Linux/OSX environment:

```
1 groovyConsole -cp src/main/groovy/:src/main/resources/ example.groovy
```

Compilation

```
1 groovyc
```

To take advantage of the JDK 7 invoke-dynamic instruction, use the `--indy` flag[1]. This also works with the `groovy` command.

Invoke-dynamic helps the compiler improve the performance of things like duck-typing, meta-programming, and method-missing calls.

Shell

```
1 groovysh
```

The groovy shell can be used to execute Groovy code in an interactive command shell.

Try it out!

Documentation

```
groovydoc
```

This tool generates documentation from your groovy code.

Groovy uses the same comment syntax as Java including the conventions for documenting code.

```
/** This is a documentation comment. */
/* This is not */
// This is a one-line comment.
```

1. Available in Groovy 2.0 and above.↩

GDK

The GDK (Groovy Development Kit) provides a number of helper methods, helper operators, utilities, and additional classes.

Some of these are methods added to every Java class, like `each` and some are more obscure.

Collections

Groovy adds tons of helpful methods that allow easier manipulation of collections:

- `sort` – Sorts the collection (if it is sortable)
- `findAll` – Finds all elements that match a closure.
- `collect` – An iterator that builds a new collection.
- `inject` – Loops through the values and returns a single value.
- `each` – Iterates through the values using the given closure.
- `eachWithIndex` – Iterates through with two parameters: a value and an index.
- `find` – Finds the first element that matches a closure.
- `findIndexOf` – Finds the first element that matches a closure and returns its index.
- `any` – True if any element returns true for the closure.
- `every` – True if all elements return true for the closure.
- `reverse` – Reverses the ordering of elements in a list.
- `first` – Gets the first element of a list.
- `last` – Returns the last element of a list.
- `tail` – Returns all elements except the first element of a list.

Spread

The spread operator can be used to access the property of every element in a collection. It can be used instead of “collect” in many cases. For example, you could print the name of every dragon thusly:

```
1 dragons*.name.each { println it }
```

GPath

GPath is something like X-path in Groovy. Thanks to the support of property notation for both lists and maps, Groovy provides syntactic sugar making it really easy to deal with nested collections, as illustrated in the following examples:

```
1 def listOfMaps = [['a': 11, 'b': 12], ['a': 21, 'b': 22]]
2 assert listOfMaps.a == [11, 21] //GPath notation
3 assert listOfMaps*.a == [11, 21] //spread dot notation
4
5 listOfMaps = [['a': 11, 'b': 12], ['a': 21, 'b': 22], null]
6 assert listOfMaps*.a == [11, 21, null] // caters for null values
7 assert listOfMaps*.a == listOfMaps.collect { it?.a } //equivalent notation
```

```
8 // But this will only collect non-null values
9 assert listOfMaps.a == [11,21]
```

IO

The GDK helps you a lot with input/output (IO).

Files

The GDK adds several methods to the File class to ease reading and writing files.

```
1 println path.toFile().text
```

A `getText()` method is added to the `File` class that simply reads the whole file.

```
1 new File("books.txt").text = "Modern Java"
```

Here we are using the `setText` method on the `File` class which simply writes the file contents.

For binary files you can also use the `bytes` property on File:

```
1 byte[] data = new File('data').bytes
2 new File('out').bytes = data
```

In case you want to use an InputStream or Reader or the corresponding OutputStream or Writer for output you have the following methods:

```
1 new File('dragons.txt').withInputStream {in -> }
2 new File('dragons.txt').withReader {r -> }
3 new File('dragons.txt').withOutputStream {out ->}
4 new File('dragons.txt').withWriter {w -> }
```

Lastly you can use the `eachLine` method to read each line of a file. For example:

```
1 new File('dragons.txt').eachLine { line->
2     println "$line"
3 }
4 //OR
5 new File('dragons.txt').eachLine { line, num ->
6     println "Line $num: $line"
7 }
```

In all of these cases, Groovy takes care of closing the I/O resource even if an exception is thrown.

Print out a multi-line file and then read it back in and print out the lines.

URLs

The GDK makes it extremely simple to execute a URL.

The following Java code opens an HTTP connection on the given URL (“http://google.com” in this case), reads the data into a byte array, and prints out the resulting text.

```
URL url = new URL("http://google.com");
InputStream input = (InputStream) url.getContent();
ByteArrayOutputStream out = new ByteArrayOutputStream();
int n = 0;
byte[] arr = new byte[1024];

while (-1 != (n = input.read(arr)))
    out.write(arr, 0, n);

System.out.println(new String(out.toByteArray()));
```

However, in Groovy this also can be reduced to one line (leaving out exceptions):

```
println "http://google.com".toURL().text
```

A `toURL()` method is added to the `String` class and a `getText()` method is added to the `URL` class in Groovy.

Use Groovy to download your favorite web site and see if you can parse something from it.

Ranges

The `Range` is a built-in type in Groovy. It can be used to perform loops, in switch cases, extracting substrings, and other places. Ranges are generally defined using the syntax `start..end`.

Ranges come in handy for traversing using the `each` method and “for-loops”:

```
(1..4).each {print it} //1234
for (i in 1..4) print i //1234
```

A `case` statement was demonstrated in an earlier chapter such as the following:

```
case 12..30: // Range 12 to 30
```

This only works if the value given to switch statement is the same type as the Range (Integer in this case).

You can use Ranges to extract substrings from a String using the `getAt` syntax. For example:

```
def text = 'learning groovy'
println text[0..4] //learn
println text[0..4,8..-1] //learn groovy
```

Negative numbers count down from the last element of a Collection or String. So `-1` equates to the last element.

You can also use Ranges to access elements of a List:

```
1 def list = ['hank', 'john', 'fred']
2 println list[0..1] //[hank, john]
```

You can define a Range to be exclusive of the last number by using the `..<` operator. For example another way to print `1234` would be the following:

```
1 (1..<5).each {print it} //1234
```

Attempt to use a variable in a Range. Do you need to surround the variable with parentheses?

Utilities

The GDK adds several utility classes such as ConfigSlurper, Expando, and ObservableList/Map/Set.

ConfigSlurper

ConfigSlurper is a utility class for reading configuration files defined in the form of Groovy scripts. Like with Java *.properties files, ConfigSlurper allows a dot notation. It also allows for nested (Closure) configuration values and arbitrary object types.

```
 1 def config = new ConfigSlurper().parse('''
 2     app.date = new Date()
 3     app.age  = 42
 4     app {
 5         name = "Test${42}"
 6     }
 7 ''')
 8
 9 def properties = config.toProperties()
10
11 assert properties."app.date" instanceof String
12 assert properties."app.age" == '42'
13 assert properties."app.name" == 'Test42'
```

Expando

The Expando class can be used to create a dynamically expandable object.

```
1 def expando = new Expando()
2 expando.name = { -> 'Draco' }
3 expando.say = { String s -> "${expando.name} says: ${s}" }
4 expando.say('hello') // Draco says: hello
```

Use meta-programming to alter some class's `metaClass` and then print out the class of the `metaClass`. Is it the `Expando` class?

ObservableList/Map/Set

Groovy comes with observable lists, maps and sets. Each of these collections trigger `PropertyChangeEvent` (from the `java.beans` package) when elements are added, removed

or changed. Note that a PropertyChangeEvent does not only signal an event has occurred, it also holds information on the property name and the old/new value of a property.

For example, here's an example using `ObservableList` and printing out the class of each event:

```
def list = new ObservableList()
def printer = {e -> println e.class}
list.addPropertyChangeListener(printer)
list.add 'Harry Potter'
list.add 'Hermione Granger'
list.remove(0)
println list
```

This would result in the following output:

```
class groovy.util.ObservableList$ElementAddedEvent
class java.beans.PropertyChangeEvent
class groovy.util.ObservableList$ElementAddedEvent
class java.beans.PropertyChangeEvent
class groovy.util.ObservableList$ElementRemovedEvent
class java.beans.PropertyChangeEvent
[Hermione Granger]
```

This can be useful for using the *Observer* pattern on collections.

Use an ObservableMap and a PropertyChangeListener to reject null values from being added to the Map.

Coming from Java

Since most readers are already familiar with Java, it would be helpful to compare common Java idioms with the Groovy equivalent.

Default Method Values

One thing that might surprise you coming from Java is in Groovy you can provide default values for method parameters. For example, let's say you have a `fly` method with a parameter called `text`:

```
1 def fly(String text = "flying") {println text}
```

This would essentially create two methods behind the scenes (from a Java stand-point):

```
1 def fly() {println "flying"}
2 def fly(String text) {println text}
```

This can work with any number of parameters as long as the resulting methods do not conflict with other existing methods.

Equals, Hashcode, etc.

One of the tedious tasks you must often do in Java is create both an equals and hashCode method for a class. For this purpose, Groovy added the `@EqualsAndHashCode` annotation. Simply add it to the top of your class (right before the word `class`) and you're done.

Likewise, you often want to create a constructor for all of the fields of a class. For this, Groovy has `@TupleConstructor`. It simply uses the ordering of the definitions of your fields to also define a constructor.

Also theres the `@ToString` annotation for automatically creating a toString() method.

Finally, if you want to have all of these things on your class just use the `@Canonical` annotation.

```
1 import groovy.transform.*
2 @Canonical class Dragon {def name}
3 println new Dragon("Smaug")
4 // prints: Dragon(Smaug)
5 assert new Dragon("").equals(new Dragon(""))
```

Create your own class with multiple properties using the above annotations.

Regex Pattern Matching

Groovy greatly simplifies using a pattern to match text using regex (Regular Expresssions).

Where in Java you must use the `java.util.regex.Pattern` class, create an instance and then create a Matcher, in Groovy this can all be simplified to one line.

By convention you surround your regex with slashes. This allows you to use special regex syntax without using the tedious double back-slash. For example to determine if something is an email (sort of):

```
1 def isEmail = email ==~ /[\w.]+@[\w.]+/
```

The equivalent code in Java would be:

```
1 Pattern patt = Pattern.compile("[\\w.]+@[\\w.]+");
2 boolean isEmail = patt.matches(email);
```

There's also an operator for creating a `Matcher` in Groovy:

```
1 def email = 'mailto:adam@email.com'
2 def mr = email =~ /[\w.]+@[\w.]+/
3 if (mr.find()) println mr.group()
```

This allows you to find regular expressions inside strings and to get sub-groups from a regex.

Create a better regex for validating email in Groovy.

Missing Java Syntax

Due to the nature of Groovy's syntax and some additions to Java's syntax over the years, Groovy is "missing" a few things. However, there are other ways to do the same things.

Arrays can be somewhat difficult to work with in Groovy because the Java syntax for creating an array of values does not compile. For example, the following would not compile in Groovy:

```
1 String[] array = new String[] {"foo", "bar"};
```

You should instead use the following syntax (if you must use an array):

```
1 String[] array = ['foo', 'bar'].toArray()
```

For a long time the `for (Type item : list)` syntax was not supported in Groovy. Instead you have two options: using the `in` keyword or using the `each` method with a closure. For example:

```
1 // for (def item : list)
2 for (item in list) doStuff(item)
3 list.each { item -> doStuff(item) }
```

Semi-colon Optional

Since the semi-colon is optional in Groovy this can sometimes cause line-ending confusion when you're used to Java. Usually this is not a problem, but when calling multiple methods in a row (using a fluent API for example) this can cause problems. In this case you need to end each line with a non-closed operator, like a dot for example.

For example, let's take an arbitrary fluent API:

```
class Pie {
  def bake() { this }
  def make() { this }
  def eat() { this }
}
def pie = new Pie().
      make().
      bake().
      eat()
```

If you were to use the typical Java syntax, it would case a compilation error in Groovy:

```
def pie = new Pie() //Groovy interprets end of line
      .make() // huh? what is this?
```

Where are Generics?

Groovy supports the syntax of generics but does not enforce them by default. For this reason you won't see a lot of generics in Groovy code. For example, the following would work fine in Groovy:

```
List<Integer> nums = [1, 2, 3.1415, 'pie']
```

However, Groovy will enforce generics if you add the `@StaticCompile` or `@TypeChecked` annotation to your class or method. For example:

```
import groovy.transform.*
@CompileStatic
class Foo {
    List<Integer> nums = [1, 2, 3.1415] //error
}
```

This would cause a compilation error "Incompatible generic argument types. Cannot assign java.util.List <java.lang.Number> to: java.util.List <Integer>" Since `3.1415` becomes a `java.math.BigDecimal` in Groovy, the generic type of the List is automatically determined to be `java.lang.Number`.

Groovy Numbers

Which leads us to decimal numbers use `BigDecimal` by default in Groovy. This allows you do math without rounding errors.

If you would like to use `double` or `float`, simply follow your number with `d` or `f` respectively (as you can do in Java also). For example:

```
def pie = 3.141592d
```

Try out multiplying different number types and find out the class of the result.

Boolean-resolution

Since Groovy is very similar to Java, but not Java, it's easy to get confused by their differences. A couple of these confusions are *boolean-resolution* (also called "Groovy truth") and the *Map syntax sugar*.

Groovy is much more liberal in what it accepts in a boolean expression. For example, the empty-string and zero are considered `false`. So, the following prints out "true" four times:

```
1 if ("foo") println("true")
2 if (!"") println("true")
3 if (42) println("true")
4 if (! 0) println("true")
```

This is sometimes referred to as "Groovy Truth".

Map Syntax

Groovy syntax-sugar for Maps allows you use String keys directly, which is often very helpful. However, this can cause confusion when attempting to get the class-type of a map using Groovy's property-accessor syntax sugar (`.class` refers to the key-value, not `getClass()`). So you should use the `getClass()` method directly.

This can also cause confusion when trying to use variables as keys. In this case you need to surround the variables with parentheses. For example:

```
1 def foo = 1
2 def bar = 2
3 def map = [(foo): bar]
```

Without the parentheses `foo` would resolve to the string `"foo"`.

Summary

In this chapter you've learned the following Groovy features:

- You can provide default method values.
- Various annotations that simplify life in the `groovy.transform` package.
- How regular expressions are built-in to Groovy.
- You should define arrays differently than Java.
- How to use unclosed operations when writing a multi-line statement.
- Groovy uses BigDecimal by default for non-integer numbers.
- Groovy truth.
- Using variable keys in the map syntax.

Advanced Groovy

Beyond the basics, Groovy is a rich tapestry of language features. It can be used dynamically or statically-typed, your choice. It supports functional programming constructs, including first-class functions, currying, and more. It has multiple-inheritance, type inference, and meta-programming.

Groovy Design Patterns

Design patterns are a great way to make your code functional, readable, and extensible. There are some patterns that are easier and require less code in Groovy compared to Java.

Strategy Pattern

Imagine you have three different methods for finding totals:

```
def totalPricesLessThan10(prices) {
     int total = 0
     for (int price : prices)
             if (price < 10) total += price
     total
}
def totalPricesMoreThan10(prices) {
     int total = 0
     for (int price : prices)
             if (price > 10) total += price
     total
}
def totalPrices(prices) {
     int total = 0
     for (int price : prices)
             total += price
     total
}
```

A lot of code is duplicated in this case. There's only one small thing that changes for each of these methods. In Groovy you can use a closure parameter instead of three different methods so you can have the following:

```
def totalPrices(prices, selector) {
      int total = 0
      for (int price : prices)
              if (selector(price)) total += price
      total
}
```

Now you have a method, `totalPrices(prices, selector)` where selector is a closure. Also, you can put the closure outside of the method parameters in a method call if it's the last parameter. So you can call the above method three different ways to achieve the desired results:

```
totalPrices(prices) { it < 10 }
totalPrices(prices) { it > 10 }
totalPrices(prices) { true }
```

This not only makes your code more concise, it's also easier to read and extend.

Meta-programming

Groovy meta-programming means you can add methods to any class at runtime. This allows you to add helper methods to commonly used classes to make your code more concise and readable.

Meta-Class

For example, let's say you're writing a `javax.servlet.Filter` and you get and set session attributes a lot. You could do the following:

```
1 HttpSession.metaClass.getAt = { key -> delegate.getAttribute(key) }
2 HttpSession.metaClass.putAt = {
3     key, value -> delegate.setAttribute(key, value)
4     }
```

This allows the following syntax:

```
1 def session = request.session
2 session['my_id'] = '123'
3 def id = session['my_id']
```

Categories

The *Category* is one of the many meta-programming techniques available in Groovy. A Category is a class that can be used to add functionality onto existing classes.

To make a Category, you create some static methods that have at least one parameter of a particular type (eg. Integer). When the category is used that type (of the first parameter) appears to have those methods. The object on which the method is called is used as the parameter.

For example, Groovy has TimeCategory for manipulating dates and times. This lets you add and subtract any arbitrary length of time. For example:

```
1 import groovy.time.TimeCategory
2 def now = new Date()
3 println now
4 use(TimeCategory) {
5     nextWeekPlusTenHours = now + 1.week + 10.hours - 30.seconds
6 }
7 println nextWeekPlusTenHours
```

In this case, TimeCategory adds a bunch of methods to the Integer class. For example, some of the method signatures look like the following:

```
1 static Duration getDays(Integer self)
2 static TimeDuration getHours(Integer self)
3 static TimeDuration getMinutes(Integer self)
4 static DatumDependentDuration getMonth(Integer self)
5 static TimeDuration getSeconds(Integer self)
```

Create your own Category class and then put it on GitHub.

Method Missing

In Groovy you can intercept missing methods using the `methodMissing` method as follows:

```
1 def methodMissing(String name, args)
```

Next you can intercept, cache and invoke the called method (GORM in Grails uses this for its query functions). For example:

```
def methodMissing(String name, args) {
      impl = { /* your code */ }
      getMetaClass()."$name" = impl
      impl()
}
```

This implements the missing functionality and then adds it to the current class's metaClass so that future calls go directly to the implementation instead of the `methodMissing` method.

Delegation

Delegation is when a class has methods that directly call (method signature identical) methods of another class. This is hard in Java because it is difficult and time consuming to add methods to a class.

This is much easier with the new `@Delegate` annotation in Groovy 2.0. It's like compile-time meta-programming. It automatically adds the methods of the delegate class to the current class.

For example:

```
public class Person {
      def eatDonuts() { println("yummy") }
}

public class RoboCop {
      @Delegate final Person person

      public RoboCop(Person person) { this.person = person }
      public RoboCop() { this.person = new Person() }

      def crushCars() {
              println("smash")
      }
}
```

Although RoboCop does not have an `eatDonuts()` method, all of the methods of Person are added to RoboCop and delegated to `person`. This allows for the following usage:

```
def person = new RoboCop()
person.eatDonuts()
person.crushCars()
```

Use `@Delegate` on a List property and use it to make a List that cannot have elements removed.

DSLs

Groovy has many features that make it great for writing DSLs (Domain Specific Languages):

- Closures with delegates.
- Parenthesis and dots (.) are optional.
- Ability to add methods to standard classes using Category and Mixin transformations.
- The ability to override many operators (plus, minus, etc.)
- The methodMissing and propertyMissing methods.

Domain Specific Languages can be useful for many different purposes such as allowing domain experts to read and write code, or clarify the meaning of business logic.

Delegate

Within Groovy you can take a block of code (a closure) as a parameter and then call it using a local variable as a delegate. For example, imagine you have the following code for sending SMS texts:

```
1 class SMS {
2       def from(String fromNumber) {
3               // set the from
4       }
5       def to(String toNumber) {
6               // set the to
7       }
8       def body(String body) {
9               // set the body of text
10      }
11      def send() {
12              // send the text.
13      }
14 }
```

In Java, you'd need to use this the following way:

```
1 SMS m = new SMS();
2 m.from("555-432-1234");
3 m.to("555-678-4321");
4 m.body("Hey there!");
5 m.send();
```

In Groovy you can add the following static method to the SMS class for DSL-like usage:

```
1 def static send(block) {
2       SMS m = new SMS()
3       block.delegate = m
4       block()
5       m.send()
6 }
```

This sets the SMS object as a delegate for the block so that methods are forwarded to it. With this you can now do the following:

```
SMS.send {
      from '555-432-1234'
      to '555-678-4321'
      body 'Hey there!'
}
```

This removes a lot of repetition from the code.

As demonstrated, you can omit the parentheses when making a simple method call.

Overriding Operators

In Groovy you can override operators simply by naming your methods using the English word for the operator. For example `plus` for `+` and `minus` for `-`.

Operator	Method-name
+	plus
-	minus
*	multiply
/	div
%	mod
**	power
|	or
&	and
^	xor
<<	leftShift
>>	rightShift
++	next
--	previous

All of these methods have one parameter except for “next” and “previous” which have no parameters.

For example, let’s create a class called `Logic` with a boolean value and define the `and` and `or` methods.

```
class Logic {
    boolean value
    Logic(v) {this.value = v}
    def and(Logic other) {
     this.value && other.value
    }
    def or(Logic other) {
     this.value || other.value
    }
}
```

Then let’s use these methods and see if they work like you would expect:

```
def pale = new Logic(true)
def old = new Logic(false)

println "groovy truth: ${pale && old}"
println "using and: ${pale & old}"
println "using or: ${pale | old}"
```

You'll notice that using the built-in `&&` operator uses "Groovy truth" and returns true because both variables are non-null.

Next let's try out defining the `leftShift` and `minus` operators on a class:

```
class Wizards {
    def list = []
    def leftShift(person) { list.add person }
    def minus(person) { list.remove person }
    String toString() { "Wizards: $list" }
}
def wiz = new Wizards()
wiz << 'Gandolf'
println wiz
wiz << 'Harry'
println wiz
wiz - 'Harry'
println wiz
```

You can also implement map-like `putAt` and `getAt` methods as demonstrated in the next section. This allows you to use the bracket syntax, for example:

```
def value = object[parameter] // uses getAt
object[parameter] = value // uses putAt
```

Method-missing and Property-missing

As noted previously, Groovy provides a way to implement functionality at runtime via the `methodMissing` method:

```
def methodMissing(String name, args)
```

However, Groovy also provides a way to intercept missing properties that are accessed using Groovy's property syntax. Property access is implemented using `propertyMissing(String name)` (which returns a value) and property modification via `propertyMissing(String name, Object value)` (which sets the value for a property).

For example, here's an excerpt from a DSL for chemical compounds:

```
class Chemistry {
  public static void exec(Closure block) {
    block.delegate = new Chemistry()
    block()
  }
  def propertyMissing(String name) {
    def comp = new Compound(name)
    (comp.elements.size() == 1 && comp.elements.values()[0]==1) ?
      comp.elements.keySet()[0] : comp
  }
}
```

In this example, `propertyMissing` creates a new Compound object and returns either the Compound or an Element object if there is only one element in the Compound. This enables the creation of Compounds based on the name of a "missing" property. For example:

```
def c = new Chemistry()
def water = c.H2O
println water
println water.weight
```

This gets interpreted as trying to access a property named “H2O” which triggers the `propertyMissing` method.

H2O refers to the chemical composition of water, two hydrogens and one oxygen atom.

By using the static `exec` method, this DSL reaches its full potential by exposing an instance of Chemistry as a delegate to the closure, which allows the following example:

```
Chemistry.exec {
      def water = H2O
      println water
      println water.weight
}
```

This has the same effect of creating the H2O compound by calling the `propertyMissing` method of Chemistry.

The full code for Groovy Chemisty is provided on github. It provides the ability to compute atomic weights of chemical compounds and percentages by atomic weight. It includes all known elements, their names, and atomic weights.

Create a DSL in Groovy for something that interests you, be it Sports, Math, Movies, or Astrophysics.

Traits

Traits are like interfaces with default implementations and state. Traits in Groovy are inspired by Scala's traits.

Those familiar with Java 8 know that it added default methods to interfaces. Traits are similar to Java 8 interfaces but with the added ability to have state (fields). This allows more flexiblity, but should be treated with caution.

Defining Traits

A trait is defined using the trait keyword:

```
trait Animal {
    int hunger = 100
    def eat() { println "eating"; hunger-- }
    abstract int getNumberOfLegs()
}
```

As the above demonstrates, traits can have properties, methods, and abstract methods. If a class implements a trait it must implement its abstract methods.

Using Traits

To use a trait, you use the *implements* keyword. For example:

```
class Rocket {
    String name
    def launch() { println(name + " Take off!") }
}
trait MoonLander {
    def land() { println("${getName()} Landing!") }
    abstract String getName()
}
class Apollo extends Rocket implements MoonLander {
}
```

So now we can do the following:

```
def apollo = new Apollo(name: "Apollo 12")
apollo.launch()
apollo.land()
```

This would generate the following output:

```
Apollo 12 Take off!
Apollo 12 Landing!
```

Unlike super-classes, you can use multiple traits in one class. Here is such an example:

```
trait Shuttle {
    boolean canFly() { true }
    abstract int getCargoBaySize()
}
class MoonShuttle extends Rocket
    implements MoonLander, Shuttle {
```

```
7     int getCargoBaySize() { 100 }
8 }
```

This would allow you to do the following:

```
1 MoonShuttle m = new MoonShuttle(name: 'Taxi')
2 println "${m.name} can fly? ${m.canFly()}"
3 println "cargo bay: ${m.getCargoBaySize()}"
4 m.launch()
5 m.land()
```

And you would get the following output:

```
1 Taxi can fly? true
2 cargo bay: 100
3 Taxi Take off!
4 Taxi Landing!
```

See what happens when you have the same fields or methods in two different Traits and then try to mix them.

Summary

This chapter taught you about:

- What Traits are - similar to Java 8 interfaces.
- How to define a trait with fields and methods.
- How you can use multiple traits in one class.

Functional Programming

Functional Programming (FP) is a programming style that focuses on functions and minimizes changes of state (using immutable data structures). It is closer to expressing solutions mathematically rather than step-by-step instructions.

In FP, functions should be “side-effect free” (nothing outside the function is changed) and *referentially transparent* (a function returns the same value every time when given the same arguments).

FP can be seen as an alternative to the more common *imperative programming* which is closer to telling the computer the steps to follow.

Although functional-programming can be achieved in Java pre-Java-8, Java 8 enabled language-level FP-support with Lambda expressions and *functional interfaces*.

Java 8, JavaScript, Groovy, and Scala all support functional-style programming although they are not strictly FP languages.

Functions and Closures

As you might know, “functions as a first-class feature” is the basis of functional programming. *First-class feature* means that a function can be used anywhere a value could be used.

For example, in JavaScript, you can assign a function to a variable and call it:

```
1 var func = function(x) { return x + 1; }
2 var three = func(2); //3
```

Although Groovy doesn’t have first-class functions, it has something very similar: closures. A closure is simply a block of code wrapped in curly-brackets with parameters defined left of the `->` (arrow). For example:

```
1 def closr = {x -> x + 1}
2 println( closr(2) ); //3
```

If a closure has one argument it can be referenced as `it` in Groovy. For example:

```
1 def closr = {it + 1}
```

In Groovy the `return` keyword can be omitted if the returned value is the last expression.

Using Closures

When a closure is the last or only parameter to a method it can go outside of the parentheses. For example, the following defines a method that takes a list and a closure for filtering items:

```
def find(list, tester) {
    for (item in list)
        if (tester(item)) return item
}
```

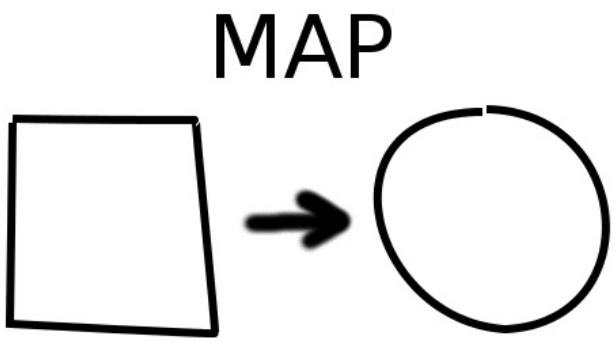
This method returns the first item in the list for which the closure returns true. Here's an example of calling the method with a simple closure:

```
find([1,2]) { it > 1 }
```

Map/Filter/etc.

Once you have mastered functions, you quickly realize you need a way to perform operations on collections (or sequences or streams) of data.

Since these are common operations, people invented *sequence operations* such as `map`, `filter`, `reduce`, etc.

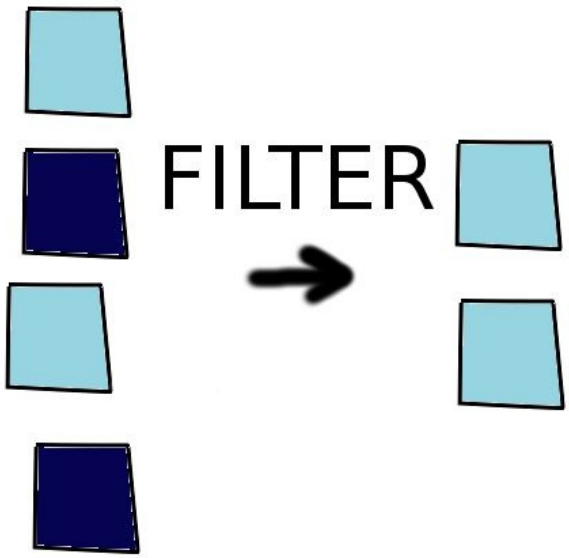


Map

Map (collect): Translates or changes input elements into something else.

```
def names = persons.collect { person -> person.name }
```

- `collect` - Iterator that builds a collection (equivalent of Scala's `map`)

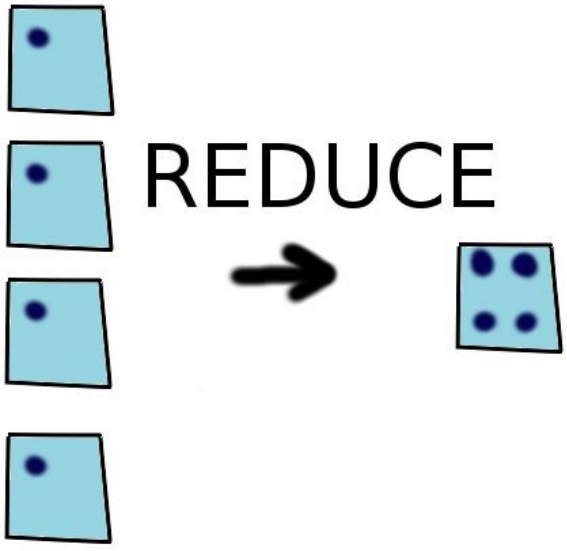


Filter

Filter (findAll): Gives you a sub-set of elements (what returns true from some *predicate* function).

```
1 def adults = persons.findAll { person -> return person.age >= 18 }
```

- `findAll` - Finds all elements that match a closure (similar to `filter` in Scala).

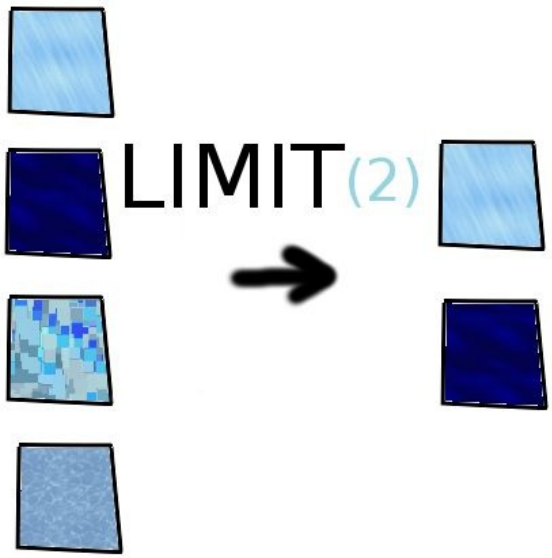


Reduce

Reduce (inject): Performs a reduction on the elements.

```
1 def totalAge = persons.inject(0) {(total, p) -> return total+p.age }
```

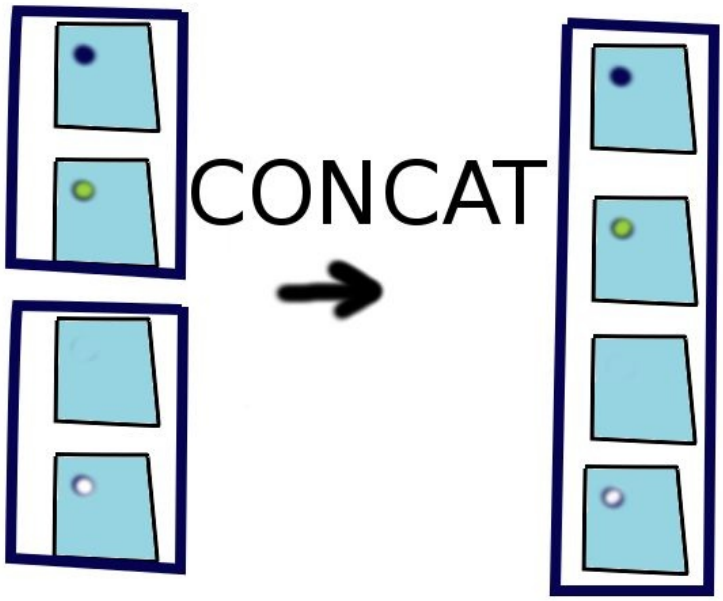
- `inject(startValue)` - Loops through the values and returns a single value (equivalent of `foldRight` in Scala).



Limit

Limit ([0..n-1]): Gives you only the first N elements.

```
1 def firstTwo = persons[0..1]
```



Concat

Concat (+): Combines two different collections of elements.

```
1 def a = [1,2,3]
2 def b = [4,5]
3 a+b
4 // Result: [1, 2, 3, 4, 5]
```

Immutability

Immutability and FP go together like peanut-butter and jelly. Although it’s not necessary, they go together nicely.

In purely functional languages the idea is that each function has no effect outside itself - no “side effects”. This means that every time you call a function it returns the same value given the same inputs.

To accommodate this behavior there are *immutable* data-structures. An immutable data-structure cannot be directly changed, but returns a new data-structure with every operation.

For example, Scala’s default Map is immutable:

```
1 val map = Map("Smaug" -> "deadly")
2 val map2 = map + ("Norbert" -> "cute")
3 println(map2) // Map(Smaug -> deadly, Norbert -> cute)
```

So in the above `map` would remain unchanged.

Each language has a keyword for defining immutable variables (values). Java has the `final` keyword for declaring immutable variables, which Groovy also respects.

```
1 public class Centaur {
2     final String name
3     public Centaur(name) {this.name=name}
4 }
5 Centaur c = new Centaur("Bane");
6 println(c.name) // Bane
7 c.name = "Firenze" //error
```

In addition to the `final` keyword, Groovy includes the `@Immutable` annotation for declaring a whole class immutable. It also adds default constructors with parameter-order, hashCode, equals, and toString methods. For example (in Groovy):

```
1 import groovy.transform.Immutable
2 @Immutable
3 public class Dragon {
4     String name
5     int scales
6 }
7 Dragon smaug = new Dragon('Smaug', 499)
8 println smaug
9 // Output: Dragon(Smaug, 499)
```

This works for simple references and primitives such as numbers and Strings but for things like lists and maps it’s more complicated. For these cases open-source immutable libraries have been developed, for example Guava for Java and Groovy.

Groovy Fluent GDK

In Groovy `findAll` and other methods are available on every object, but are especially useful for lists, sets, and ranges. The following method names are used in Groovy:

- `findAll` – Much like `filter`, it finds all elements that match a closure.
- `collect` – Much like `map`, an iterator that builds a collection.
- `inject` – Much like `reduce`, it loops through the values and returns a single value.
- `each` – Iterates through the values using the given closure.
- `eachWithIndex` – Iterates through with two parameters: a value and an index.
- `find` – Finds the first element that matches a closure.
- `findIndexOf` – Finds the first element that matches a closure and returns its index.

For example, `collect` makes it very simple to perform an operation on a list values:

```
1 def list = ['foo','bar']
2 def newList = []
3 list.collect( newList ) { it.substring(1) }
4 println newList //  [oo, ar]
```

For another example, assuming `dragons` is a List of Dragon objects:

```
1 def dragons = [new Dragon('Smaug', 499), new Dragon('Norbert', 488)]
2 String longestName = dragons.
3     findAll { it.name != null }.
4     collect { it.name }.
5     inject("") { n1, n2 -> n1.length() > n2.length() ? n1 : n2 }
```

Remember that `it` in Groovy can be used to reference the single argument of a closure.

Using the above code as a starting point, find the Dragon with the most scales.

Groovy Curry

The `curry` method allows you to pre-define values for parameters of a Closure. It takes any number of arguments and replaces parameters from left to right as you might expect. For example, given a `concat` closure, you could create a `burn` closure and a `inate` closure easily as follows:

```
1 def concat = { x, y -> return x + y }
2 // closure
3 def burn = concat.curry("burn")
4 def inate = concat.curry("inate")
```

Since you're only providing the first parameter, these closures are prepending their given text: burn prepends "burn", and inate prepends "inate". For example:

```
1 burn(" wood") // == burn wood
```

You could then use another closure called `composition` to apply the two functions to one input.

```
1 def composition = { f, g, x -> return f(g(x)) }
2 def burninate = composition.curry(burn, inate)
3 def trogdor = burninate(' all the people')
4 println "Trogdor: ${trogdor}"
5 // Trogdor: burninate all the people
```

Functional composition is an important idea in functional programming. It allows you compose functions together to create complex algorithms out of simple building blocks.

Method Handles

Method handles allow you to refer to actual methods much like they are closures. For example, given a method:

```
def breathFire(name) { println "Burninating $name!" }
```

You could later on do the following (within the same class that defined breathFire):

```
['the country side', 'all the people'].each(this.&breathFire)
```

This would pass the `breathFire` method to the `each` method as a closure, causing the following to be printed:

```
Burninating the country side
Burninating all the people
```

Create a method with multiple parameters and attempt to call it using a method-handle. Does it work as expected?

Tail Recursion

In Groovy 1.8 the trampoline method was introduced for a closure to use the tail recursion optimization. This allows closure invocations to be invoked sequentially instead of stacked to avoid a StackOverFlowError and improve performance.

Starting in Groovy 2.3 you can use trampoline for recursive methods as well the `@TailRecursive` AST transformation. Simply annotate a tail-recursive method with `@TailRecursive`and Groovy does the rest. For example:

```
import groovy.transform.*
@TailRecursive
long totalPopulation(list, total = 0) {
  if (list.size() == 0)
    total
  else
    totalPopulation(list.tail(), total + list.first().population)
}
```

On a Groovy List, the `tail` method returns the list without the first element and `first` returns just the first element.

This would take a list of objects with a `population` property and get the sum of them. For example, let's make a `City` class, use a Range to create a bunch of them, and then use our `totalPopulation` method:

```
@Canonical class City {int population}
def cities = (10..1000).collect{new City(it)}
totalPopulation(cities)
// 500455
```

This demonstrates how tail recursion can be used in a functional style as an alternative to iteration.

Tail recursion goes well with immutability because no direct modification of variables is necessary in tail recursion.

Summary

In this chapter, you should have learned about:

- Functions as a first-class feature as Closures.
- Map/Filter/Reduce as `collect/findAll/inject`.
- Immutability and how it relates to FP.
- Various features supporting FP in Groovy.
- Method Handles in Groovy
- Tail recursion optimization.

Groovy GPars

Gpars began as a separate project bringing many concurrency abstractions to Groovy, but was bundled in Groovy 1.8 and beyond.

It includes parallel map/reduce, Actors, and many other concurrency models.

Parallel Map Reduce

You perform parallel map/reduce with the GPars library in the following way:

```
GParsPool.withPool {
    // a map-reduce functional style (students is a Collection)
    def bestGpa = students.parallel
        .filter{ s -> s.graduationYear == Student.THIS_YEAR }
        .map{ s -> s.gpa }
        .max()
}
```

The static method `GParsPool.withPool` takes in a closure and augments any Collection with several methods (using Groovy's Category mechanism). The `parallel` method actually creates a ParallelArray (JSR-166) from the given Collection and uses it with a thin wrapper around it.

Actors

The *Actor design pattern* is a useful pattern for developing concurrent software. In this pattern each Actor executes in its own thread and manipulates its own data. The data cannot be manipulated by any other thread. Messages are passed between the Actors to cause them to change the data. You can also make stateless Actors.

> When data can only be changed by one thread at a time we call it *thread-safe*.

```
import groovyx.gpars.actor.Actor
import groovyx.gpars.actor.DefaultActor

class Dragon extends DefaultActor {
    int age

    void afterStart() {
        age = new Random().nextInt(1000)
    }
    void act() {
        loop {
            react { int num ->
              if (num > age)
              reply 'too old'
              else if (num < age)
              reply 'too young'
              else {
                reply 'you guessed right!'
                terminate()
              }
            }
```

```
        }
    }
}
// Guesses the age of the Dragon
class Guesser extends DefaultActor {
    String name
    Actor server
    int myNum

    void act() {
        loop {
          myNum = new Random().nextInt(10)
          server.send myNum
          react {
            switch (it) {
              case 'too old': println "$name: $myNum was too old"; break
              case 'too young': println "$name: $myNum was too young"; break
              case 'you win': println "$name: I won $myNum"; terminate(); break
            }
          }
        }
    }
}

def master = new Dragon().start()
def player = new Guesser(name: 'Guesser', server: master).start()

//this forces main thread to live until both actors stop
[master, player]*.join()
```

The Groovy Ecosystem

There are many different frameworks built on top of Groovy that make up the Groovy ecosystem.

Groovy Awesomeness

Web and UI Frameworks

Grails
: Web-framework inspired by Ruby-on-Rails; has at least 800 plugins.

Griffon
: Swing UI, command-line very similar to grails: `create-app cool -archetype=jumpstart`

vert.x
: A framework for asynchronous application development. Not strictly a Groovy project but you can use it. It's currently an Eclipse Foundation project.

Ratpack
: A toolkit for web applications on the JVM and RESTful web-services (microservices).

Cloud Computing Frameworks

Gaelyk
: An abstraction over GAE (Google App Engine); has an emerging plugin system.

Caelyf
: Born in 2011, Apache 2 licensed framework for CloudFoundry; similar to gaelyk.

Build Frameworks

Gradle
: A Groovy DSL for building projects. Uses `build.gradle`.

Gant
: Like Ant in Groovy. Born in 2006; now in maintenance mode. Was used by Grails & Griffon.

Testing Frameworks/Code Analysis

Easyb
: BDD - behaviour driven development, human readable.

Spock
: DSL testing framework. Around since 2007. Uses strings as method names. You can use data-tables for test input. `@Unroll(String)` - works like JUnit theories "unrolled".

Codenarc
: Static code analysis for Groovy. Around since 2009. Has plugins for Grails and Griffon.

GContracts
: Enforces contracts in your code. `@Requires, @Ensures`

Concurrency

GPars
: A multi-threading framework for Groovy. It has a fork/join abstraction, actors, STM, and more (it actually comes bundled with Groovy).

RxGroovy
: This is a Groovy adapter to RxJava - a library for composing asynchronous and event-based programs using observable sequences for the JVM.

Others

gvm
: The Groovy enVironment Manager (GVM); now SDKMAN (The Software Development Kit Manager), very cool. Allows you to manage multiple versions of several Groovy and non-Groovy applications including Groovy itself.

lazybones
: A simple project creation tool that uses packaged project templates. This can be installed using gvm/sdkman.

Gradle

Gradle is a groovy based DSL for building projects.

The Gradle website describes it as follows:

> Gradle combines the power and flexibility of Ant with the dependency management and conventions of Maven into a more effective way to build. Powered by a Groovy DSL and packed with innovation, Gradle provides a declarative way to describe all kinds of builds through sensible defaults. -gradle.org

Projects and Tasks

Each Gradle build is composed of one or more projects and each project is composed of tasks.

The core of the gradle build is the `build.gradle` file (which is called the *build script*).

Tasks can be defined by writing `task` then a task-name followed by a closure. For example:

```
task upper << {
      String someString = 'test'
      println "Original: $someString"
      println "Uppercase: " + someString.toUpperCase()
}
```

Tasks can contain any groovy code, but you can take advantage of existing Ant tasks; for example:

```
ant.loadfile(srcFile: file, property: 'x') //loads file into x
ant.checksum(file: file, property: "z") // put checksum into z
println ant.properties["z"] //accesses ant property z
```

The above code would load a file into ant-property “x”, save the file’s checksum in ant-property “z”, and then print out that checksum.

Much like in Ant, a task can depend on other tasks, which means they need to be run before the task. You simply call `dependsOn` with any number of task-names as arguments. For example:

```
task buildApp {
      dependsOn clean, installApp, processAssets
}
```

You can also use the following alternative syntax:

```
task buildApp(dependsOn: [clean, installApp, processAssets])
```

Use the << if your task contains the actual code you want the task to run. You are adding a Closure to the task, not configuring the task. Otherwise the code in the task is always run by Gradle, not just when you invoke the task.

Once you've defined your tasks, you run them by invoking `gradle <task_name>` at the command line. There are some built in tasks. For example, to list all available tasks invoke:

```
1 gradle tasks
```

Plugins

Gradle core has very little built-in, but it has powerful plugins to allow it to be very flexible.

A plugin can do one or more of the following:

- Add tasks to the project (e.g. compile, test)
- Pre-configure added tasks with useful defaults.
- Add dependency configurations to the project.
- Add new properties and methods to existing type via extensions.

We're going to concentrate on building Groovy-based projects, so we'll be using the groovy plugin (however, Gradle is not limited to Groovy projects!):

```
1 apply plugin: 'groovy'
```

This plugin uses Maven's conventions. For example, it expects to find your production source code under `src/main/groovy` and your test source code under `src/test/groovy`.

Configuring a Task

Once you've added some plugins, you might want to configure some of the properties of a task for your purposes.

For example you might want to specify a version of gradle for the gradle `wrapper` task:

```
1 task wrapper(type: Wrapper) {
2     gradleVersion = '2.2.1'
3 }
```

The properties available depend on what task you are configuring.

Extra Configuration

To provide extra properties within your Gradle build file use the `ext` method. You can define any arbitrary values within the closure and they will available throughout your project.

You can also apply properties from other Gradle build files. For example:

```
ext {
    apply from: 'props/another.gradle'
    myVersion = '1.2.3'
}
```

The properties defined within this closure can be used in your tasks.

Maven Dependencies

Every Java project tends to rely on many open-source projects to be built. Gradle builds on Maven so you can easily include your dependencies using a simple DSL, like in the following example:

```
apply plugin: 'java'

sourceCompatibility = 1.7

repositories {
     mavenLocal()
     mavenCentral()
}

dependencies {
     compile 'com.google.guava:guava:14.0.1'
     compile 'org.bitbucket.dollar:dollar:1.0-beta3'
     testCompile group: 'junit', name: 'junit', version: '4.+'
     testCompile "org.mockito:mockito-core:1.9.5"
}
```

This build script uses `sourceCompatibility` to define the Java source code version of `1.7` (which is used during compilation). Next it tells Maven to use the local repository first (mavenLocal), then Maven Central.

In the `dependencies` block this build script defines two dependencies for the `compile` scope and two for `testCompile` scope. Jars in the `testCompile` scope are only using by tests, and won't be included in any final products.

The line for JUnit shows the more verbose style for defining dependencies.

You can also specify your own Maven repository by calling `maven` with a closure supplying the appropriate parameters (at least a url). For example (in the `repositories` section):

```
maven { url "https://oss.sonatype.org/content/repositories/snapshots/" }
maven { url = "$nexus/content/groups/public"
      credentials {
              username 'deployment'
              password deploymentPassword
      }
}
```

The second example demonstrates using a secured maven repository. It also demonstrates using the variables `nexus` and `deploymentPassword` which could (probably should) be stored in a `gradle.properties` file.

Gradle Properties

The `gradle.properties` file allows you to specify gradle properties and other properties available to your build script. For example you can specify JVM arguments and if you want to use the Gradle daemon:

```
org.gradle.daemon=true
org.gradle.jvmargs=-Xms128m -Xmx512m
```

You could also specify build specific values that you don't want to keep in your versioning system (such as nexus credentials for example).

Multiproject builds

A mulitproject build can include any number of sub-projects that are built together. Each sub-project should be put in directory under a single top directory where the name of each directory is the name of the sub-project.

Separate builds in a multi-project build may depend on one another. You can express dependencies on any number of subprojects using the following syntax:

```
dependencies {
  compile project(':subproject1')
  compile project(':subproject2')
}
```

The top-level `build.gradle` file of multiproject should look something like the following:

```
allprojects {
  apply plugin: "groovy"
}
project(":subproject1") {
  dependencies {}
}
project(":subproject2") {
  dependencies {}
}
```

File Operations

Since Gradle evaluates tasks before actually executing them, you should generally not use `java.util.File` for defining files. Instead, Gradle provides a number of built in methods and tasks. For example, you should use the `file` method for single files and the `files` method to define a collection of files.

```
outputDir = file("libs/x86")
```

There's also a `fileTree` method for recusively listing a directory of files. For example, you can depend on all files under `lib` directory in the following way:

```
dependencies {
      compile fileTree('lib')
}
```

To copy files from one place to another you should use the `Copy` task. Here's a good example:

```
task copyImages(type: Copy) {
      from 'assets'
      into 'build/images'
      include '**/*.jpg'
      exclude '**/*test*'
}
```

The above task would copy all images that end with `.jpg` in the `assets` directory into the `build/images` directory excluding any files containing the word `test`.

Exploring

Remember you can also easily list properties of an object in Groovy using `.properties`. This can help you explore available Gradle properties at runtime. For example:

```
1 task test {
2       println sourceSets.main.properties
3 }
```

Completely Groovy

Remember that all of Groovy's goodness is available in Gradle. For example, the following sets the encoding option for two tasks using the "star-dot" notation:

```
1 [compileJava, compileTestJava]*.options*.encoding = 'UTF-8'
```

Explore Gradle by making your own `build.gradle` and trying out everything above.

Summary

This chapter hopefully taught you about the following about Gradle:

- How to create tasks
- How to use plugins
- How to specify Dependencies.
- What gradle.properties is all about.
- How to create multiproject builds.
- Using built-in methods for file operations.

Online Documentation

Gradle has a huge online user-guide available online at gradle.org.

Grails

What is Grails?

Grails is a web-framework for Groovy that follows the example of *Ruby on Rails* to be an opinionated web framework with a command-line tool that gets things done really fast. Grails uses convention over configuration to reduce configuration overhead.

Grails lives firmly in the Java ecosystem and is built on top of technologies like Spring and Hibernate. Grails also includes an Object-Relational-Mapping (ORM) framework called *GORM* and has a large collection of plugins.

Different versions of Grails can be very different, so care needs to be taken when upgrading your Grails application, especially with major versions (2.4 to 3.0 for example).

Quick Overview of Grails

After installing Grails[1], you can create an app by running the following on the command-line:

```
1 $ grails create-app
```

Then, you can run commands like `create-domain-class` and `generate-all` to create you application as you go. Run `grails help` to see the full list of commands available.

Grails applications have a very specific project structure. The following is a simple break-down of *most of* that structure:

- **grails-app** - The Grails-specific folder.
  - conf - Configuration, such as the DataSource script and Bootstrap class.
  - controllers - Controllers with methods for index/create/edit/delete or anything else.
  - domain - Domain model; classes representing your persistent data.
  - i18n - Message bundles.
  - jobs - Any scheduled jobs you might have go here.
  - services - Back-end services where your back-end or “business” logic goes.
  - taglib - You can very easily define your own tags for use in your GSP files.
  - views - Views of MVC; typically these are GSP files (HTML based).
- **src** - Any utilities or common code that doesn’t fit anywhere else.
  - java - Java code.
  - groovy - Groovy code.
- **web-app**
  - css - CSS style-sheets.
  - images - Images used by your web-application.
  - js - Your JavaScript files.

- WEB-INF - Spring's `applicationContext.xml` goes here.

To create a new domain (model) class, run the following:

```
$ grails create-domain-class
```

It's a good idea to include a package for your domain-classes (like `example.Post`).

A domain class in Grails also defines its mapping to the database. For example, here's a domain class representing a blog post (assuming User and Comment were already created):

```
class Post {
    String text
    int rating
    Date created = new Date()
    User createdBy

    static hasMany = [comments: Comment]

    static constraints = {
     text(size:10..5000)
    }
}
```

The static `hasMany` field is a map which represents one-to-many relationships in your database. Grails uses Hibernate in the background to create tables for all of your domain classes and relationships. Every table gets an `id` field for the primary key by default.

To have Grails automatically create your controller and views, run:

```
$ grails generate-all
```

Grails will ask if you want to overwrite existing files if they exist. So be careful when using this command.

When you want to test your app, you simply run:

```
$ grails run-app
```

When you're ready to deploy to an application container (eg. Tomcat), you can create a "war" file by typing:

```
$ grails war
```

Plugins

The Grails ecosystem now includes over one-thousand plugins. To list all of the plugins, simply execute:

```
$ grails list-plugins
```

When you've picked out a plugin you want to use, execute the following (with the plugin name and version):

```
$ grails install-plugin [NAME] [VERSION]
```

This will add the plugin to your project. If you decide to uninstall it, simply use the `uninstall-plugin` command.

REST in Grails

Groovy includes some built-in support for XML, such as the XMLSlurper. Grails also includes converters for simply converting objects to XML or JSON or vice versa.

```
1 import grails.converters.JSON
2 import grails.converters.XML
3
4 class BookController {
5         def getBooks = {
6                 render Book.list() as JSON
7         }
8         def getBooksXML = {
9                 render Book.list() as XML
10        }
11 }
```

For REST services that service multiple formats, you can use the built-in `withFormat` in Grails. So the above would become the following:

```
1 def getBooks = {
2       withFormat {
3               json { render list as JSON }
4               xml { render list as XML }
5       }
6 }
```

Then Grails would decide which format to use based on numerous inputs, the simplest being the extension of the URL such as '.json' or the request's "Accept" header.

For *using* web-services in Grails, there's a rest plugin available.

Short History of Grails

What follows is a brief history of features added to Grails starting with `2.0`.

Grails 2.0

There have been a lot of great changes in Grails 2.0:

- Grails docs are better.
- Better error page - shows code that caused the problem.
- H2 database console in development mode (at the URI `/dbconsole`)
- Grails 2.0 supports Groovy 1.8.
- Runtime reloading for typed services, domain classes, `src/groovy` and `src/java`
- Run any command with `-reloading` to dynamically reload it.
- Binary plugins (jars).
- Better scaffolding - HTML 5 compliant (mobile/tablet ready).
- PageRenderer and LinkGenerator API for services.
- Servlet 3.0 async API supported; events plugin; platform core.
- Resources plugin integrated into core.
- Plugins for gzip, cache, bundling (install-plugin cached-resources, zipped-resources).
- New tags: `img, external, javascript`

- The jQuery plugin is now the default JavaScript library installed into a Grails application.
- A new `date` method has been added to the params object to allow easy, null-safe parsing of dates: `def val = params.date('myDate', 'dd-MM-yyyy')`

Various GORM improvements:

- Support for `DetachedCriteria` and new `findOrCreate` and `findOrSave` Methods.
- GORM now supports: MongoDB, riak, Cassandra, neo4j, redis, Amazon SimpleDB
- New compile-time checked query DSL (`where` with a closure): avg, sum, subqueries, .size(), etc.
- Multiple scoped data sources.
- Added database-migration plugin for updating production Databases.

Grails 2.1

- Grails' Maven support has been improved in a number of significant ways. For example, it is now possible to specify plugins within your pom.xml file.
- The grails command now supports a `-debug` option which will startup the remote debug agent.
- Installs the cache plugin by default.
- In Grails 2.1.1 domain classes now have static methods named `first` and `last` to retrieve the first and last instances from the datastore.

Grails 2.2

- Grails 2.2 supports Groovy 2.
- Adds new functionality to criteria queries to provide access to Hibernate's SQL projection API.
- Supports forked JVM execution of the Tomcat container in development mode.
- Includes improved support for managing naming conflicts between artifacts provided by an application and its plugins.

Grails 2.3

- Improved Dependency Management using the same library used by Maven (Aether) by default.
- Includes a new data binding mechanism which is more flexible and easier to maintain than the data binder used in previous versions.
- All major commands can now be forked into a separate JVM, thus isolating the build path from the runtime and test paths.
- Grails' REST support has been significantly improved.
- Grails' Scaffolding feature has been split into a separate plugin (includes support for generating REST controllers, Async controllers, and Spock unit tests).
- Includes new Asynchronous Programming APIs that allow for asynchronous processing of requests and integrate seamlessly with GORM.

- Controllers may now be defined in a namespace which allows for multiple controllers to be defined with the same name in different packages.

Grails 2.4

- Grails 2.4 comes with Groovy 2.3
- Uses Hibernate 4.3.5 by default (Hibernate 3 is still available as an optional install).
- The Asset-Pipeline replaces Resources to serve static assets.
- Now has great support for static type checking and static compilation. The `GrailsCompileStatic` annotation (from the `grails.compiler` package) behaves much like the `groovy.transform.CompileStatic` annotation and provides special handling for Grails.

Grails 3

Grails 3 represents a huge refactoring of Grails. The public API is now located in the `grails` package and everything has been redone to use traits. Also each new project features an Application class with a traditional static void main.

Grails 3 comes with Groovy 2.4, Spring 4.1, and Spring Boot 1.2 and the build system now uses Gradle instead of the Gant-based system.

Among other changes, the use of filters has been deprecated and should be replaced with interceptors. To create a new interceptor, use the following command:

```
grails create-interceptor MyInterceptor
```

An interceptor contains the following methods:

```
boolean before() { true }
boolean after() { true }
void afterView() {}
```

- The `before` method is executed before a matched action. The return value determines whether the action should execute, allowing you to cancel the action.
- The `after` method is executed after the action executes but prior to view rendering. Its return value determines whether the view rendering should execute.
- The `afterView` method is executed after view rendering completes.

Grails 3 supports built in support for Spock/Geb functional tests using the `create-functional-test` command.

Testing

Grails supports unit testing with a mixin approach.

Annotations:

- `@TestFor(X)`: Specifies the class under test.
- `@Mock(Y)`: Creates a Mock for the given class.

For tests GORM provides an in-memory mock database that uses a `ConcurrentHashMap`.

You can create tests for Tag libraries, command objects, URL-Mappings, XML & JSON, etc.

Cache Plugin

- You can add the `@Cacheable` annotation on Service or Controller methods.
- cache tags – `cache:block`, `cache:render`
- cache-ehcache, -redis, -gemfire
- Cache configuration DSL.

Grails Wrapper

The Grails wrapper was added in Grails 2.0. You can create the wrapper using the following command:

```
1 $ grails wrapper
```

The above produces `grailsw` and `grailsw.bat` (for *nix and Windows respectively).

The Grails wrapper is helpful when multiple people are working on a project. The wrapper scripts will actually download and install grails when run. Then you can send these scripts to anyone or keep them in your repository.

Cloud

Grails is supported by the following cloud providers:

- CloudFoundry
- Amazon
- Heroku

Only an Overview

This has only been a brief overview of Grails. Many books have been written about Grails and how to use it. For more information on using Grails, please visit grails.org.

Create your own Grails app and then deploy it in the Cloud.

Groovy Mailgun

Learning a bunch of theory is nice, but sometimes it’s helpful to see a real-life example. For this purpose, let’s implement a Grails service for sending emails from an application running in the cloud.

Let's use the Mailgun with Groovy on Heroku. Although you might think sending emails is easy, it's not if you don't want your email to end up in the SPAM folder. Mailgun provides a SaaS (Software as a service) for sending and receiving emails with free accounts available. To make things interesting we're going to use Mailgun's REST api.

First of all, you need to import a bunch of classes and declare your service and the `sendEmail` method:

```
import org.apache.http.*
import org.apache.http.entity.*
import org.apache.http.client.*
import org.apache.http.client.methods.*
import org.apache.http.impl.client.*
import org.apache.http.params.*
import org.codehaus.groovy.grails.plugins.codecs.URLCodec

/** Uses MailGun to send emails. */
class EmailService {

     static API_KEY = System.env.MAILGUN_API_KEY

    def sendEmail(to, subject, text) {
```

Second, you need to add the encodeAsURL method to Object's metaClass. You'll use this later for encoding the parameters.

```
final codec = URLCodec.newInstance()
Object.metaClass."encodeAsURL" = { -> codec.encode(delegate) }
```

Then define your API url and parameters:

```
def url = 'https://api.mailgun.net/v2/your-app.mailgun.org/messages'
def params = [from: "you@email.com", to: to, subject: subject, text: text]
```

Next, create the *DefaultHttpClient* and *HttpPost* with credentials and headers:

```
def httpclient = new DefaultHttpClient()
def post = new HttpPost(url);
def creds = new UsernamePasswordCredentials('api', API_KEY)
post.addHeader(BasicScheme.authenticate(creds, "US-ASCII", false));
post.addHeader("Content-Type", "application/x-www-form-urlencoded");
```

Lastly, encode the parameters, join them, and put them in the post.

```
def list = []
params.each {k,v -> list << "${k}=${v.encodeAsURL()}" }
post.entity = new StringEntity(list.join('&'))
def response = httpclient.execute(post);
response
```

That's it, you're done! Now you can send emails easily (Mailgun's free service comes with up to 300 emails/day).

Using other REST services in the cloud (such as Twilio) with Groovy is very similar.

The full gist of EmailService is available on github.

1. This overview is based on Grails 2.1.4, but the basics should remain the same for all versions of Grails.↩

Spock

Spock is a testing framework for Java and Groovy applications. The Spock website has this to say about Spock:

> What makes it stand out from the crowd is its beautiful and highly expressive specification language. Thanks to its JUnit runner, Spock is compatible with most IDEs, build tools, and continuous integration servers. Spock is inspired from JUnit, RSpec, jMock, Mockito, Groovy, Scala, Vulcans, and other fascinating life forms.

Introduction

The basic structure of a test class in Spock is a class that extends `Specification` and has multiple methods with Strings for names.

Spock processes the test code and allows you to use a simple groovy syntax to specify tests.

Each test is composed of labeled blocks of code with labels like “when”, “then”, and “where”. The best way to learn Spock is with examples.

A Simple Test

Let’s start by recreating a simple test from the chapter on JUnit:

```
1 def "toString yields the String representation"() {
2       def array = ['a', 'b', 'c'] as String[]
3       when:
4       def arrayWrapper = new ArrayWrapper<String>(array);
5       then:
6       arrayWrapper.toString() == '[a, b, c]'
7 }
```

As shown above, assertions are simply groovy conditional expressions. If the above “==” expression returns false, the test will fail and Spock will give a detailed printout to explain why it failed.

In the absence of any “when” clause, you can use the “expect” clause instead of “then”; for example:

```
1 def "empty list size is zero"() {
2       expect: [].size() == 0
3 }
```

Mocking

Mocking interfaces is extremely easy in Spock[1]. Simply use the `Mock` method, as shown in the following example (where `Subscriber` is an interface):

```
class APublisher extends Specification {
  def publisher = new Publisher()
  def subscriber = Mock(Subscriber)
```

Now `subscriber` is a mocked object. You can implement methods simply using the overloaded >> operator as shown below.

```
def "can cope with misbehaving subscribers"() {
    subscriber.receive(_) >> { throw new Exception() }

    when:
    publisher.send("event")
    publisher.send("event")

    then:
    2 * subscriber.receive("event")
}
```

Expected behavior is described by using a number or range times (`*`) the method call as shown above.

The under-score (_) is treated like a wildcard (much like in Scala).

Lists or Tables of Data

Much like how JUnit has DataPoints and Theories, Spock allows you to use lists or tables of data in tests.

For example,

```
def "subscribers receive published events at least once"() {
    when: publisher.send(event)
    then: (1.._) * subscriber.receive(event)
    where: event << ["started", "paused", "stopped"]
}
```

Above, the overloaded << operator is used to provide a list for the `event` variable. Although it is a List here, anything that is iterable could be used.

Ranges

The range `1.._` here means “one or more” times. You can also use `_..3`, for example, to mean “three or less” times.

Tabular formatted data can be used as well. For example:

```
def "length of NASA mission names"() {
      expect:
      name.size() == length

      where:
      name       | length
      "Mercury"  | 7
      "Gemini"   | 6
      "Apollo"   | 6
}
```

In this case, the two columns (`name` and `length`) are used to substitute the corresponding variables in the expect block. Any number of columns can be used.

Expecting Exceptions

Use the `thrown` method in the `then` block to expect a thrown Exception.

```
def "peek on empty stack throws"() {
    when: stack.peek()
    then: thrown(EmptyStackException)
}
```

You can also capture the thrown exception by simply assigning it to `thrown()`. For example:

```
def "peek on empty stack throws"() {
    when: stack.peek()
    then:
    Exception e = thrown()
    e.toString().contains("EmptyStackException")
}
```

Conclusion

As you can see, Spock makes tests more concise and easy to read, and, most importantly, makes the intentions of the test clear.

1. You can also Mock classes, but it requires including the cglib jar as a dependency.↩

Ratpack

At its core, Ratpack enables asynchronous, stateless HTTP applications. It is built on Netty, the event-driven networking engine. Unlike some other web framework, there is no expectation that one thread handles one request. Instead, you are encouraged to handle blocking operations in a way that frees the current thread thus allowing high performance.

Ratpack can be used to make responsive, RESTful microservices, although it's not a requirement.

> *REST* stands for REpresentational State Transfer. It was designed in a PhD dissertation and has gained some popularity as the new web-service standard. At the most basic level in REST each CRUD operation is mapped to an HTTP method (GET, POST, PUT, etc.).

Unlike Grails and other popular web frameworks, Ratpack aims not to be a framework, but instead a set of libraries (as it says above). Although it's a lean set of libraries, Ratpack comes packed with support for JSON, websockets, SSE (Server Sent Events), SSL, SQL, logging, Dropwizard Metrics, newrelic, health checks, hystrix and more.

Ratpack uses Guice by default for DI (dependency injection).

Ratpack is written in Java (8) and you can write Ratpack applications in pure Java, but we are focusing on the Groovy side.

Script

In the simplest form you can create a ratpack application using only a Groovy script. For example:

```
@Grab('io.ratpack:ratpack-groovy:1.1.1')

import static ratpack.groovy.Groovy.ratpack

ratpack {
  handlers {
    handler {
      response.send "Hello World!"
    }
  }
}
```

Gradle

For production systems you should use a Gradle build. Ratpack has its own Gradle plugin which you can use as follows:

```
buildscript {
  repositories {
    jcenter()
  }
  dependencies {
    classpath 'io.ratpack:ratpack-gradle:1.1.1'
  }
}

apply plugin: 'io.ratpack.ratpack-groovy'

repositories {
  jcenter()
}
```

Using the ratpack-gradle plugin you can run tasks such as `distZip`, `distTar`, `installApp`, and `run` which create a zip distribution file, tar distribution file, install the ratpack application locally, and run the application respectively.

The `run` task is very useful for test driving your application. After invoking `gradle run` you should see the following output:

```
[main] INFO ratpack.server.RatpackServer - Starting server...
[main] INFO ratpack.server.RatpackServer - Building registry...
[main] INFO ratpack.server.RatpackServer - Ratpack started (development) for htt\
p://localhost:5050
```

Ratpack Layout

- build.gradle - The Gradle build file.
- **src**
  - main/groovy - Where you put general Groovy classes.
  - main/resources - Where you put static resources such as Handlebars templates.
  - ratpack - Contains `Ratpack.groovy` which defines your ratpack handlers and bindings and ‘ratpack.properties’.
  - ratpack/public - Any public files like HTML, JavaScript, and CSS.
  - ratpack/templates - Holds your Groovy-Markup templates (if you have any).

Handlers

Handlers are the basic building block for Ratpack. They form something like a pipeline or [“Chain of Resposiblity”](). Multiple handlers can be called per request, but one must return a response. If none of your handlers are matched, a default handler returns a 404 status code.

```
import static ratpack.groovy.Groovy.ratpack

ratpack {
  handlers {
    all() { response.headers.add('x-custom', 'x'); next() }
    get("foo") {
        render "foo handler"
    }
    get(":key") {
        def key = pathTokens.key
        render "{\"key\": \"$key\"}"
    }
    files { dir "public" }
  }
}
```

The first handler `all()` is uses for every HTTP request and adds a custom header. It then calls `next()` so the next matching handler gets called. The next handler that matches a given HTTP request will be used to fulfill the request with a response.

The `pathTokens` map contains all parameters passed in the URL as specified in the matching path pattern, as in `:key`, by prefixing the ':' to a variable name you specify.

You can define a handler using a method corresponding to any one of the HTTP methods:

- get
- post
- put
- delete
- patch
- options

To accept multiple methods on the same path, you should use the `path` handler and `byMethod` with each method handler inside it. For example:

```
1   path("foo") {
2       byMethod {
3           post() { render 'post foo'}
4           put() { render 'put foo'}
5           delete() { render 'delete foo'}
6       }
7   }
```

Rendering

There are many ways to render a response. The "grooviest" ways are using the `groovyMarkupTemplate` or `groovyTemplate` methods.

For example, here's how to use the `groovyMarkupTemplate`:

```
1 import ratpack.groovy.template.MarkupTemplateModule
2 import static ratpack.groovy.Groovy.groovyMarkupTemplate
3 import static ratpack.groovy.Groovy.ratpack
4 ratpack {
5   bindings {
6     module MarkupTemplateModule
7   }
8   handlers {
9     get(":key") {
10        def key = pathTokens.key
11        render groovyMarkupTemplate("index.gtpl", title: "$key")
12    }
13    files { dir "public" }
14  }
15 }
```

This allows you to use the Groovy Markup language and by default it looks in the `ratpack/templates` directory for markup files. For example, your "index.gtpl" might look like the following:

```
1 yieldUnescaped '<!DOCTYPE html>'
2 html {
3   head {
4     meta(charset:'utf-8')
5     title("Ratpack: $title")
6     meta(name: 'apple-mobile-web-app-title', content: 'Ratpack')
7     meta(name: 'description', content: '')
```

```
    meta(name: 'viewport', content: 'width=device-width, initial-scale=1')
    link(href: '/images/favicon.ico', rel: 'shortcut icon')
  }
  body {
    header {
      h1 'Ratpack'
      p 'Simple, lean &amp; powerful HTTP apps'
    }
    section {
      h2 title
      p 'This is the main page for your Ratpack app.'
    }
  }
}
```

Groovy Text

If you prefer to use a plain old text document with embeded groovy (much like a GString) you can use the `TextTemplateModule`:

```
import ratpack.groovy.template.TextTemplateModule
import static ratpack.groovy.Groovy.groovyTemplate
import static ratpack.groovy.Groovy.ratpack
ratpack {
  bindings {
    module TextTemplateModule
  }
  handlers {
    get(":key") {
        def key = pathTokens.key
        render groovyTemplate("index.html", title: "$key")
    }
  }
}
```

Then create a file named “index.html” in the “src/main/resources/templates” directory with the following content:

```
<html><h1>${model.title}</h1></html>
```

It supplies a `model` Map to your template which contains all of the parameters you supply.

Handlebars and Thymeleaf

Ratpack also supports Handlebars and Thymeleaf templates. Since these are static resource, you should put these templates in the “src/main/resources” directory.

You will first need to include the appropriate ratpack project in your Gradle build file:

```
runtime 'io.ratpack:ratpack-handlebars:1.1.1'
runtime 'io.ratpack:ratpack-thymeleaf:1.1.1'
```

To use Handlebars, include the `HandlebarsModule` and render handlebar templates as follows:

```
import ratpack.handlebars.HandlebarsModule
import static ratpack.handlebars.Template.handlebarsTemplate
import static ratpack.groovy.Groovy.ratpack
ratpack {
  bindings {
    module HandlebarsModule
  }
  handlers {
    get("foo") {
        render handlebarsTemplate('myTemplate.html', title: 'Handlebars')
    }
```

```
  }
}
```

Create a file named “myTemplate.html.hbs” in the “src/main/resources/handlebars” directory with handlebar content. For example:

```
<span>{{title}}</span>
```

Thymeleaf works in a similar way:

```
import ratpack.thymeleaf.ThymeleafModule
import static ratpack.thymeleaf.Template.thymeleafTemplate
import static ratpack.groovy.Groovy.ratpack
ratpack {
  bindings {
    module ThymeleafModule
  }
  handlers {
    get("foo") {
        render thymeleafTemplate('myTemplate', title: 'Thymeleaf')
    }
  }
}
```

The requested file should be named “myTemplate.html” and be located in the “src/main/resources/thymeleaf” directory of your project. Example content:

```
<span th:text=\"${title}\"/>
```

JSON

Integration with the Jackson JSON marshalling library provides the ability to work with JSON as part of ratpack-core.

The ratpack.jackson.Jackson class provides most of the Jackson related functionality. For example, to render json you can use `Jackson.json`:

```
import static ratpack.jackson.Jackson.json
ratpack {
  bindings {
  }
  handlers {
    get("user") {
      render json([user: 1])
    }
  }
}
```

The Jackson integration also includes a Parser for converting JSON request bodies into objects. The `Jackson.jsonNode()` and `Jackson.fromJson(Class)` methods can be used to create parseable objects to be used with the `parse()` method.

For example, the following handler would parse a JSON string representing a Person object and render the Person’s name:

```
post("personNames") {
  render( parse(fromJson(Person.class)).map {it.name} )
}
```

`Context.parse` returns a `Promise`.

Bindings

Bindings in ratpack make objects available to the handlers. If you are familiar with Spring, ratpack uses a *Registry* which is similar to the application context in Spring. It can also be thought of as a simple map from class-types to instances. Ratpack-groovy uses Guice by default, although other direct-injection frameworks can be used (or none at all).

Instead of a formal plugin system, reusable functionality can be packaged as modules. You can create your own modules to properly decouple and organize your application into components. For example you might want to create a *MongoModule* or a *JdbcModule*. Alternatively, you could break your application into services. Either way, the registry is where you put them.

Anything in the registry can be automatically used in a handler and gets wired in by class-type from the registry. For an example of bindings in action:

```
bindings {
  bindInstance(MongoModule, new MongoModule())
  bind(DragonService, DefaultDragonService)
}
handlers {
  get('dragons') { DragonService dService ->
    dService.list().then { dragons ->
      render(toJson(dragons))
    }
  }
}
```

Blocking

Blocking operations should be handled by using the “blocking” API. A blocking operation is anything that is IO-bound such as querying the database.

The `Blocking` class is located at `ratpack.exec.Blocking`. For example, the following handler calls a method on the database:

```
get("deleteOlderThan/:days") {
  int days = pathTokens.days as int
  Blocking.get { database.deleteOlderThan(days) }
          .then { int i -> render("$i records deleted") }
}
```

You can chain multiple blocking calls using the `then` method. The result of the previous closure is passed as the parameter to the next closure (an int above).

Promise

Ratpack makes this possible by using its own implementation of a Promise. `Blocking.get` returns a `ratpack.exec.Promise`.

Ratpack handles the thread scheduling for you and then joins with the original computation thread. This way you can rejoin with the original HTTP request thread and return a result.

```
get("deleteOlderThan/:days") {
  int days = pathTokens.days as int
  int result
  Blocking.get { database.deleteOlderThan(days) }
          .then { int count -> result = count }
  render("$result records deleted")
}
```

If no return value is required, use the `Blocking.exec` method.

Configuration

Any self-respecting web application should allow configuration to come from multiple locations: the application, the file-system, environment variables, and system properties. This is a good practice in general, but especially for cloud-native apps.

Ratpack includes a built-in configuration API. The `ratpack.config.ConfigData` class allows you to layer multiple sources of configuration. Using the `of` method with a passed closure allows you to define a factory of sorts for configuration from JSON, YAML, and other sources.

First, you define your configuration classes. These class properties will define the names of your configuration properties. For example:

```
class Config {
    DatabaseConfig database
}
class DatabaseConfig {
    String username = "root"
    String password = ""
    String hostname = "localhost"
    String database = "myDb"
}
```

In this case your Json configuration might look like:

```
{
    "database": {
        "username": "user",
        "password": "changeme",
        "hostname": "myapp.dev.company.com"
    }
}
```

Later, in the `binding` declaration, add the following to bind the configuration defined by the above classes:

```
def configData = ConfigData.of { d ->
    json(getResource("/config.json")).
    yaml(getResource("/config.yml")).
    sysProps().
    env().build()
}
bindInstance(configData.get(Config))
```

Each declaration overrides the previous declarations. So in this case the order would be: “class definition”, `config.json`, `config.yml`, system-properties, then environment

variables. This way you could override properties at runtime.

System property and environment variable configuration must be prefixed with the `ratpack.` and `RATPACK_` prefixes accordingly. For example, the `hostname` property above would be `ratpack.database.hostname` as a system property.

Testing

Ratpack includes many test fixtures for aiding your unit, functional, and integration tests. It assumes you'll be using Spock to test your application.

First, if you're using the ratpack gradle plugin, simply add the following dependencies:

```
dependencies {
    testCompile ratpack.dependency('test')
    testCompile "org.spockframework:spock-core:0.7-groovy-2.0"
    testCompile 'cglib:cglib:2.2.2'
    testCompile 'org.objenesis:objenesis:2.1'
}
```

The `cglib` and `objenesis` dependencies are needed for object mocking.

Second, add your Spock tests under the "src/test/groovy" directory using the same package structure as your main project. A simple test might look like the following:

```
package myapp.services
import spock.lang.Specification

class MyServiceSpec extends Specification {
    void "default service should return Hello World"() {
        setup:
        "Set up the service for testing"
        def service = new MyService()
        when:
        "Perform the service call"
        def result = service.doStuff()
        then:
        "Ensure that the service call returned the proper result"
        result == "Hello World"
        "Shutdown the service when this feature is complete"
        service.shutdown()
    }
}
```

Ratpack enables your functional tests by running your full application within a test environment using `GroovyRatpackMainApplicationUnderTest`. For example, the following test would test the text rendered by your default handler:

```
package myapp
import ratpack.groovy.test.GroovyRatpackMainApplicationUnderTest
import spock.lang.Specification

class FunctionalSpec extends Specification {
    void "default handler should render Hello World"() {
        setup:
        def aut = new GroovyRatpackMainApplicationUnderTest()
        when:
        def response = aut.httpClient.text
        then:
        response == "Hello World!"
        cleanup:
        aut.close()
    }
}
```

Merely calling “text” on the httpClient invokes a GET request. However, more complex requests can be invoked using `requestSpec`. For example, to test a specific response is returned based on the “User-Agent” header:

```
void "should properly render for v2.0 clients"() {
    when:
    def response = aut.httpClient.requestSpec { spec ->
        spec.headers.'User-Agent' = ["Client v2.0"]
    }.get("api").body.text
    then:
    response == "V2 Model"
}
```

Summary

This chapter hopefully taught you about the following:

- How to get started with Ratpack.
- Using handlers.
- How to use bindings as a plugin architecture and for decoupling your modules.
- How to render using Groovy Markup, Groovy Text, Thymeleaf, or Handlebars templates.
- How to render and parse JSON.
- Doing Blocking operations.
- Configuring a Ratpack app.
- Testing a Ratpack app.

For more information you can check out the Ratpack API.

Also, you should go buy the book Learning Ratpack by Dan Woods.

Appendixes

This is where everything goes that doesn’t fit anywhere else.

Java/Groovy[1]

Feature	Java	Groovy
Public Class	public class	class
Loops	for(Type it : c){…}	c.each {…}
Lists	List list = asList(1,2,3);	def list = [1,2,3]
Maps	Map m = …; m.put(x,y);	def m = [x: y]
Function Def.	void method(Type t) {}	def method(t) {}
Mutable Value	Type t	def t
Immutable Value	final Type t	final t
Null safety	(x == null ? null : x.y)	x?.y
Null replacement	(x == null ? “y” : x)	x ?: “y”
Sort	Collections.sort(list)	list.sort()
Wildcard import	import java.util.*;	import java.util.*
Var-args	(String… args)	(String… args)
Type parameters	Class<T>	Class<T>
Concurrency	Fork/Join	GPars

No Java Analogue

Feature	Groovy
Default closure arg.	it
Default value	def method(t = “yes”)
Add method to object	t.metaClass.method = {}
Auto-delegate	@Delegate
Extension methods	Categories or Traits
Rename import	import java.util.Vector as Vect

Tricks

Feature	Groovy
Range	def range = [a..z]
Slice	def slice = list[0..3]
<< Operator	list << addMeToList
Cast operation	def dog = [name: “Fido”, speak:{println “woof”}] as Dog
GString	def gString = “Dog’s name is ${dog.name}”

1. Version 1.4 of this cheat-sheet.↩

Resources

- Java Tutorials
- Java Docs
- Groovy Docs
- Spock Docs
- Gradle Docs
- Grails Docs
- Ratpack Docs
- JavaOne (by going to Tools - Content Catalog)
- StackOverflow