Groovy Basics

What is Groovy?

An agile and **dynamic language** for the **Java Virtual Machine**

Builds upon the strengths of Java but has **additional power features** inspired by languages like Python, Ruby and Smalltalk

Key Features

- Dynamically typed
- Intuitive - it tries to do the right thing by default
- Interoperability with Java -> Groovy is Java
  - compiles to Java byte code
  - uses standard Java data types
  - uses and enhances standard Java libraries
  - use standard Java build tools like Maven, Ant, Gradle
  - use standard test tools like JUnit, Cobertura
  - standard packaging - .class files in jar, war, ear

Quick Diff from Java

- Dynamic
  - Optional typing
  - Scripts can be run without pre-compilation
  - Methods, properties can be added to classes at runtime
- Relaxed Script-like Syntax
  - Optional semicolons, parentheses, returns
- Added Features
  - Properties
  - Native syntax for lists, maps and ranges
  - Support embedded expressions in Strings
  - Additional operators: Null safe, Elvis, etc
  - Closures
  - Operator overloading
  - Meta Object Protocol
  - Native support for regular expressions

Data types

- def a = 1 // Integer
- def b = 1.0 // BigDecimal
- def c = true // Boolean
- def d = [1,2,3] // ArrayList
- def e = [x:1, y:2] // LinkedHashMap

All data types are objects. Primitives can be used but are autoboxed.

Strings

java.lang.String literal uses single quotes

```
def s = ‘Here is a simple string’


// the plus operator performs string concatenation
def firstName = 'Steve'
def username = 'srogers'
println 'hello ' + firstName + ' (' + username + ')'
```

=> hello Steve (srogers)

Groovy Strings aka GStrings

- GString literals specified with double quote
- supports embedded expressions
- braces are optional for simple expressions
- triple quote for multi-line GStrings (handy for XML/json)

```
// using dollar sign and braces
println "hello ${employee.firstName} (${employee.username})"
```

=> hello Steve (srogers)

```
// braces are optional for simple object.property notation
println "hello $employee.firstName ($employee.username)"
```

=> hello Steve (srogers)

Groovy is Object Oriented

- Code is organized into *classes*

- Classes have
  - methods (procedures, functions)
  - properties (fields)
  - getters, setters automatically created

- Grails will map database data into Groovy classes
  - tables == classes
  - columns == properties

A Simple Groovy Class

```
class Employee {
        String username
        String password
        String firstName
        String middleInitial
        String lastName
        String email
        BigDecimal salary
        boolean enabled
        boolean accountExpired
        boolean accountLocked
        boolean passwordExpired
}
```

Constructing New Objects

new **creates a new object:**

```
def steveRogers = new Employee()
steveRogers.firstName = 'Steve'
steveRogers.lastName = 'Rogers'
steveRogers.username = 'srogers'
steveRogers.password =  'CaptainAmerica'
```

Groovy name-argument constructor **:**

```
def steveRogers = new Employee( username: 'srogers’,
                                password: 'CaptainAmerica',
                                firstName: 'Steve',
                                lastName: 'Rogers')
```

Creating Methods - similar to Java

- same as functions, procedures
- go inside of a class

```
scope returnType methodName(zeroOrMoreArgs) {
    method body
}
```

```
public String toString() {
    return "first: $firstName last: $lastName username: $username"
}
```

Groovier Methods

- scope is optional - default is public
- return type is optional - use ***def*** keyword for dynamically typed return value
- types are optional for method arguments
- return keyword is optional; value of last

```
// Java version
public String createGreeting(String greeting) {
    return greeting + firstName;
}
// Groovy
def createGreeting(greeting) {
    "$greeting $firstName"
}
```

Using a Method

```
def steveRogers = new Employee( username: 'srogers',  password: 'CaptainAmerica',  firstName: 'Steve', lastName: 'Rogers')


// invoke a method on an object, just like Java
println steveRogers.toString()
```

first: Steve last: Rogers username: srogers

```
// call a method, passing in a method argument,  just like java
println steveRogers.createGreeting('hello')
```

hello Steve

```
// Groovy - optional parens
println steveRogers.createGreeting ‘hello’
```

hello Steve

Conditionals

if statement works like Java except that

- == is equivalent to Java .equals()
- Groovy Truthiness
  - null, zero (0), empty strings, empty lists, & empty maps are all *false*
  - everything else evaluates to *true*

Operator overloading

`a == b`	`a.equals(b)`
`a + b`	`a.plus(b)`
`a << b`	`a.leftShift(b)`
`if (a)`	`if(a.asBoolean())`
`switch(a) { case(b) : }`	`b.isCase(a)`
`a > b`	`a.compareTo(b) > 0`
`a <=> b`	`a.compareTo(b)`

- All operators are method calls under the covers
- Methods can be overridden
- The big win for business applications is operators for BigDecimal

http://groovy.codehaus.org/Operator+Overloading

More Operators

- Ternary operator
  - x ? "x is true" : "x is false"
- Elvis operator x ?: y
  - same as x ? x : y
- Null safe operator ?.
  - person?.address?.zipCode

Regular Expressions

- Built-in operators and functions that use regular expressions
  - assert 'cheese123' **=~** 'cheese' //any matches
  - assert 'cheese123' **==~** 'cheese[0-9]*' //matches entire string
- No escaping when using "slashy" String syntax
  - assert '\\backslash' == /\backslash/
  - assert 'cheese123' ==~ /cheese\d*/

Collections: Lists

Groovy Lists

```
// create a new list, java.util.ArrayList
def employees = [ ]

// add an employee to that list
employees.add( new Employee(firstName: 'Steve'))

// use operator overloading to add another employee
employees << new Employee(firstName: 'Peter')
```

Groovy Lists: size(), indexed access

```
// .size() works on everything in Groovy, unlike Java which .length, .size(), etc...
def employees = [
        new Employee(firstName: 'Steve'),
        new Employee(firstName: 'Bruce'),
        new Employee(firstName: 'Peter') ]
println employees.size()
=> 3

// array-like indexed access into the list
println employees[0]
=> Steve
```

Closures

- Similar to code blocks or lambda expressions in other languages
- Can be called like a method
- Code as data
  - assign code blocks to variables
  - **pass them into methods**
  - return them from methods

```
def shout = {String s -> println s.toUpperCase()}
shout 'Steve'
=> STEVE
```

Looping with Closures

Most interesting methods on a collection take a *closure*.

each allows you to *iterate* over a list:

```
employees.each { employee ->
        println employee.firstName
}

// each element in a collection referenced by it - but cannot nest it
// the default iterator is great for one liners
employees.each { println it.firstName }
```

Looping with For Statement

```
def employees = [
    new Employee(firstName: 'Steve'),
    new Employee(firstName: 'Bruce'),
    new Employee(firstName: 'Peter')]

for (employee in employees) {
        // do something with employee
}
```

Collect Values from a List

```
employees = [
    new Employee(firstName: 'Steve'),
    new Employee(firstName: 'Bruce'),
    new Employee(firstName: 'Peter')]

def firstNames = employees.collect {it.firstName}
```

=> [Steve, Bruce, Peter]

```
// Spread operator
employees*.firstName
```

=> [Steve, Bruce, Peter]

Filter Lists - find

```
def employees = [   new Employee(firstName: 'Steve', middleInitial: 'G', lastName: 'Rogers'),
                    new Employee(firstName: 'Bruce', lastName: 'Banner'),
                    new Employee(firstName: 'Peter', lastName: 'Parker')]

// find first occurrence
employees.find { it.firstName == 'Bruce' }
=> Bruce Banner
```

Filter Lists - findAll

```
def employees = [   new Employee(firstName: 'Steve', middleInitial: 'G', lastName: 'Rogers'),
                    new Employee(firstName: 'Bruce', lastName: 'Banner'),
                    new Employee(firstName: 'Peter', lastName: 'Parker')]

// findAll returns a collection, ||  or operator
employees.findAll { it.firstName.startsWith('S') || it.lastName.startsWith('P') }
=> [ 'Steve G Rogers', 'Peter Parker']

// groovy truthiness finds all employees with a middle initial
employees.findAll { it.middleInitial }
=> [Steve G Rogers]
```

Lists - Operator Overloading

+ , **<<** : add object(s) to a collection

// call Java .add() method on an ArrayList

println ['Steve','Bruce'].add('Peter')

=> ['Steve','Bruce','Peter']

println ['Steve','Bruce'] **+** 'Peter'

=> ['Steve','Bruce','Peter']

println ['Steve','Bruce'] **<<** 'Peter'

=> ['Steve','Bruce','Peter']

println ['Steve','Bruce'].addAll(['Peter', 'Donald'])

=> ['Steve', 'Bruce', 'Peter', 'Donald']

println ['Steve', 'Bruce'] **+** ['Peter', 'Donald']

=> ['Steve','Bruce','Peter', 'Donald']

println ['Steve','Bruce'] **<<** ['Peter','Donald']

=> ['Steve','Bruce', ['Peter','Donald']]

Destructive Methods

Some methods mutate the list, others return a modified list.

```
employees = [    new Employee(firstName: 'Steve'),    new Employee(firstName: 'Bruce'), new Employee(firstName: 'Peter')]

def steve = employees[0]

// use Groovy minus operator to remove one from the list

println employees - steve

=> ['Bruce','Peter']

println employees.size()

=> 3

println employees.remove(steve)   // use Java remove() method to remove one from the list

=> ['Bruce','Peter']

println employees.size()

=> 2
```

Membership

any, **every** return a boolean for meeting membership criteria

```
def employees = [
        new Employee(firstName: 'Steve', middleInitial: 'G', lastName: 'Rogers'),
        new Employee(firstName: 'Bruce', lastName: 'Banner'),
        new Employee(firstName: 'Peter', lastName: 'Parker')]

employees.any { it.firstName == 'Steve' }
=> true
employees.every { it.middleInitial }
=> false
employees.every { it.lastName && it.firstName }
=> true
```

Sorting

- .sort() can be called on any collection
- .sort() can take a closure, so you don't need dedicated comparators
- sort(false) doesn't modify a collection, returns a new one instead

```
['Steve','Bruce','Peter'].sort()
=> ['Bruce', 'Peter','Steve']

employees.sort(false) { it.lastName }
=> ['Bruce Banner', 'Peter Parker', 'Steve Rogers']

employees
=> ['Steve Rogers', 'Bruce Banner', 'Peter Parker']
```

Groovy Basics: Lab

Lists

- constructing
- accessing
- mutating

Closures

- iterating with each
- collect, spread operator
- find, findAll
- any, every

Collections: Maps

Constructing Maps

- maps are like dictionaries or association lists in other languages
- key/value stores

```
def m = [:]
m.put('firstName', 'Steve')
m.put('lastName', 'Rogers')
println m
=>[firstName:Steve, lastName:Rogers]
def n = [firstName: 'Peter', lastName: 'Parker']
println n
=>[firstName:Peter, lastName:Parker]
```

Maps: Accessing Values

```
def m = [firstName: 'Steve', lastName: 'Rogers']
// use dot notation for accessing values with Strings as keys
m.firstName
=> Steve
// also use index notation
println m['firstName']
=> Steve
// mutate value with assignment
n = [:]
n.firstName = 'Peter'
n.lastName = 'Parker'
n['firstName'] = 'Peter'
n['lastName'] = 'Parker'
```

Maps: Iteration using .each

```
def m = ['Captain America':'Steve Rogers', 'Spider Man':'Peter Parker']

// each will automatically deconstruct the key/value pairs
m.each { alias, realName ->
   println "key: $alias  value: $realName"
}

=>
key: Captain America  value: Steve Rogers
key: Spider Man  value: Peter Parker
```

Sorting

- Maps don't have an order for elements
- .sort() can take a closure, so you don't need dedicated comparators
- sort(false) doesn't modify a collection, returns a new one instead

```
['Steve':'Captain America','Bruce':'Hulk', 'Peter': 'Spiderman'].sort()
=> ['Bruce':'Hulk', 'Peter':'Spiderman', 'Steve':'Captain America']

employeeMap.sort(false) {e1, e2 -> e1.key <=> e2.key  }
=> ['Bruce':'Hulk', 'Peter':'Spiderman', 'Steve':'Captain America']

employeeMap
=>['Steve':'Captain America','Bruce':'Hulk', 'Peter': 'Spiderman']
```

Ranges

1..10 - is a list that includes the Integers from 1 to 10

These are used in Grails for things like validating the size of a field i.e. password 6..20

```
def one_to_ten = 1..10
one_to_ten.size()
=> 10

one_to_ten.each { print it}
=>12345678910
```

Groovy Just Like Java, Except...

Equality

- Groovy == is the same as Java .equals(), value equality
- Groovy === and the ***is*** operator, object identity

Collection Class Methods

- Java methods mutate collections
- Groovy methods return a new collection

Groovy Basics: Lab

Maps

- construction using literals
- various accessors
- iterate using .each

One example combining everything learned thus far.