NPRG014 – 2025 plan

29th Sep - Groovy (syntax, scripting, functional programming) (VP)

6th Oct - Groovy (dynamic meta-programming, DSLs) (VP)

13th Oct - Groovy (static meta-programming) (VP)

20th Oct – Scala (TB)

27th Oct - Scala (TB)

3rd Nov - Scala (TB)

10th Nov - Scala (TB)

17th Nov – Bank holidays

24th Nov – Modern concurrency (VP)

1st Dec - Prototype-based languages - IO (TB)

8th Dec - Prototype-based languages - JavaScript, TypeScript (TB)

15th Dec - Backup

Criteria

A homework assignment will be given at each lecture

A solution to each homework must be submitted through the Teams project **by the start of the following lecture**

Submit **at least 8 correctly implemented homeworks**

Repository

Clone and then checkout before each lecture

https://github.com/d3scomp/NPRG014.git

Communication using Teams

Language dynamism, scripting and functional programming

Václav Pech

NPRG014 2025/2026

http://www.vaclavpech.eu

@vaclav_pech

Today's agenda

Why Groovy?

Scripting

Functional programming

- Groovy syntax and interoperability
- Language dynamism

Today's agenda

You will learn:

The common characteristics of dynamic languages

The benefits of languages with scripting support

Recap some of the fundamentals of functional programming in the context of a new language

Groovy

A JVM programming language (started 2003)

- Object-oriented
- Building on Java syntax
- Dynamic
- Dynamically-typed
- Scripting

Why Groovy

Flat learning curve

Concise, readable and expressive syntax, easy to learn for Java developers

Smooth Java integration

Seamlessly and transparently integrates and interoperates with Java and any third-party libraries

Vibrant and rich ecosystem

Web development, reactive applications, concurrency / asynchronous / parallelism library, test frameworks, build tools, code analysis, GUI building

Powerful features

Closures, builders, runtime & compile-time meta-programming, functional programming, type inference, and static compilation

Domain-Specific Languages

Flexible & malleable syntax, advanced integration & customization mechanisms, to integrate readable business rules in your applications

Scripting and testing glue

Great for writing concise and maintainable tests, and for all your build and automation tasks

They all use Apache Groovy!

Part 1

Groovy syntax and interoperability

Interoperability

Groovy and *Java* can **implement**, **extend**, **refer** and **call** each other at will.

Groovy sources compile into *.class* files

IDEs provide cross-reference support

Java

```
public class Person {
    private final String name;
    public Person(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
}
```

Groovy

```
public class Person {
    private final String name;
    public Person(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
}
```

Groovy (optional ;)

```
public class Person {
    private final String name
    public Person(String name) {
        this.name = name
    }
    public String getName() {
        return name
    }
}
```

Groovy (optional *return*)

```
public class Person {
    private final String name
    public Person(String name) {
        this.name = name
    }
    public String getName() {
        return name
    }
}
```

Groovy (optional *return*)

```
public class Person {
    private final String name
    public Person(String name) {
        this.name = name
    }
    public String getName() {
        name
    }
}
```

Groovy (*public* is default)

```
public class Person {
    private final String name
    public Person(String name) {
        this.name = name
    }
    public String getName() {
        name
    }
}
```

Groovy (*public* is default)

```
class Person {
    private final String name
    Person(String name) {
        this.name = name
    }
    public String getName() {
        name
    }
}
```

Groovy (*properties*)

```
class Person {
    private final String name
    Person(String name) {
        this.name = name
    }
    public String getName() {
        name
    }
}
```

Groovy (*properties*)

```
class Person {
    final String name
    Person(String name) {
        this.name = name
    }
}
```

Groovy (*named parameters*)

```
class Person {
    final String name
    Person(String name) {
        this.name = name
    }
}
```

Groovy > Java

```
class Person {
    final String name
}
```

Variables, constants, params

String s - a variable

def s – a variable (type inferred at run-time)

final s – a constant value (type inferred at run-time)

Intuitiveness

Equality *a == b*

Identity *a.is(b)*

() sometimes optional: *println 'Joe'*

String interpolation

```
final s = 'Hi Joe'

final s = "Hi Dave"

final s = "Hi $name"

final s = "Hi ${user.name}"

final s = """Hi Dave,

How are you?

"""
```

Numbers and primitive types

15 - integer

15G - BigInteger

1.5 - BigDecimal

1.5d - Double

All values are objects: 5.upto(10)

Clever boxing and unboxing

Properties

```
class City {
    String name
    int size
    boolean capital = false
}

City c1 = new City(name: 'Praha', size: 1200000, capital: true)
City c2 = new City(name: 'Písek', size: 25000)

print c1.name
c2.size = 25001
```

Power assert

```
assert 5 == customer.score
```

```
Exception thrown
17.2.2012 12:30:12 org.codehaus.groovy.runtime.StackTraceUtils sanitize

WARNING: Sanitizing stacktrace:

Assertion failed:

assert 5 == customer.score
         |  |        |
         |  |        4
         |  [score:4]
         false
```

Closures

```
Closure multiply = {int a, int b -> return a * b}
```

Closures

```
Closure multiply = {int a, int b -> a * b}
```

Closures

```
Closure multiply = {a, b -> a * b}
```

Closures – implicit parameter

```
def triple1 = {int number -> number * 3}

def triple2 = {number -> number * 3}

def triple3 = {it * 3}
```

Groovy is functional

```
def multiply = {a, b -> a * b}
def double = multiply.curry(2)
def triple = multiply.curry(3)

assert 4 == multiply(2, 2)
assert 8 == double(4)
assert 6 == triple(2)
```

Currying vs. Partial application

```
def multiply = {a, b → a * b}

def partial = multiply.curry(3)

def curried1 = {x → {y → multiply(x, y)}}

def curried2 = {x → multiply.curry(x)}
```

Memoize

def func = {a → longComputation(a)}

def fastFunc = func.memoize()

Closure scope

owner

delegate

this

closure.resolveStrategy =

DELEGATE_FIRST / OWNER_FIRST

DELEGATE_ONLY / OWNER_ONLY

Changing closure scope

closure.delegate = obj

- changes the closure

copy = closure.rehydrate(del, owner, this)

- clones the closure

with(obj, closure)

- clones the closure

Collections

```
final emptyList = []
final list = [1, 2, 3, 4, 5]
final emptyMap = [:]
final capitals = [cz : 'Prague', uk : 'London']

final list = [1, 2, 3, 4, 5] as LinkedList
final emptyMap = [:] as ConcurrentHashMap
```

Collections API

```
(1..10).each {println it}
2.step(10, 2) {println it}

(10..20).findAll{it%2==0}
        .collect {3*it}
        .inject(0){acc, v -> acc + v}
```

map, filter, and reduce explained with emoji 😂

```
map([🐮, 🍠, 🐔, 🌽], cook)
=> [🍔, 🍟, 🍗, 🍿]

filter([🍔, 🍟, 🍗, 🍿], isVegetarian)
=> [🍟, 🍿]

reduce([🍔, 🍟, 🍗, 🍿], eat)
=> 💩
```

(Not exhaustive) list

each (aka for loop)

collect (aka map)

inject (aka reduce)

findAll (aka filter)

sum, size, findFirst, grep, groupBy

any, every, min, max, ...

Some more operators

```
['Java', 'Groovy']*.toUpperCase()

customer?.shippingAddress?.street

return user.locale ?: defaultLocale
```

GDK = JDK + FUN

- java.util.Collection
  - each(), find(), join(), min(), max() …
- java.lang.Object
  - any(), every(), print(), invokeMethod(), …
- java.lang.Number
  - plus(), minus(), power(), upto(), times(), …

Tip: Ask *DefaultGroovyMethods* for help

Syntax enhancements

- Dynamic (duck) typing – optional!
- GDK
- Syntax enhancements
  - Properties, Named parameters
  - Closures
  - Collections and maps
  - Operator overloading
  - …

List comprehension (Python)

odd = [x for x in range(0, 100) if x % 2 !=0]

*squares = [x*x for x in odd]*

Generators (Python)

```
def fibonacci():
    a = 0
    b = 1
    yield b
    while True:
        a, b = b, a + b
        yield b


allFibs = fibonacci()
```

Part 2

Scripting

Agenda

- Scripting
- Script engine customization

Scripting

Evaluate custom Groovy code

At run-time!!!

new GroovyShell().evaluate('println(“Hi!”)')

Why scripting?

Runtime configuration

Runtime rule-engine customization

Interpretation of user scripts

Interpretation of LLM-generated code

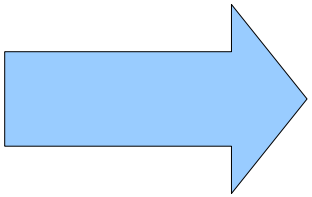
Why scripting?

Runtime configuration

Runtime rule-engine customization

Interpretation of user scripts

Interpretation of LLM-generated code



Security is key

Script customization

CompilerConfiguration

CompilationCustomizer

- ImportCustomizer
- ASTCustomizer
- SecureASTCustomizer

Functors

Dealing with wrapped data

map: ([A], f: A -> B) → [B]

map: (Maybe<A>, f: A -> B) → Maybe<B>

Functors are *mappable* (they have a **map** operation)

Monoids

Aggregating data and operations

Monoids

Aggregating data and operations

- A set of elements
- An operation that combines two elements
- An 'id' element neutral with respect to the operation
- Closure of the set with respect to the operation

1. $a + id = id + a = a$
2. $(a + b) + c = a + (b + c)$
3. $a \in M \,\&\, b \in M \Rightarrow a+b \in M$

Monoids

Reducible – any set of elements from a monoid can be reduced into a single value

reduce: ([A], f: (A, A) → A) → A

Which is not a monoid?

reduce(ints, {a, b → a+b}

*reduce(ints, {a, b → a*b}*

reduce(ints, {a, b → a-b}

reduce(ints, {a, b → a/b}

Which is not a monoid?

reduce(ints, {a, b → a+b}

*reduce(ints, {a, b → a*b}*

reduce(ints, {a, b → a-b}

reduce(ints, {a, b → a/b}

Breaks rules 1 and 2

Which is not a monoid?

reduce(ints, {a, b → a+b}

*reduce(ints, {a, b → a*b}*

reduce(ints, {a, b → a-b}

reduce(ints, {a, b → a/b}

Breaks all three rules

Monoids

class Customer {name, address, orders}

vs.

class CustData {orders, totalAmount}

Monoids

class Customer {name, address, orders}

not a monoid

vs.

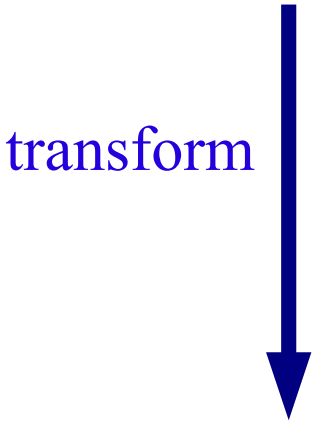
class CustData {orders, totalAmount}

a monoid

Monoids

class Customer {name, address, orders}

not a monoid



vs.

class CustData {orders, totalAmount}

a monoid

Reduce vs. Fold

m.reduce {v1, v2 → v1 + v2}

m.foldLeft(0) {acc, v → acc + v}

Composing functions

f: A → B

g: B → C

f >> g: A → C

Composing functions

f: A → B

g: B → C

f >> g: A → C

```
def f = {String s → s.size()}

def g = {Integer i → i%2==0 ? true : false}

def h = f >> g
```

Composing functions

f: A → B

g: B → C

f >> g: A → C

Not a monoid

Endofunctors

f: A → A

with composition (**>>**) and an **id()** function form a monoid

[f1, f2, f3, f4, f5, …].reduce(id, >>)

Other monoids of functions

Elements: f: String → Boolean

Other monoids of functions

Elements: f: String → Boolean

id() – returns *true/false*

Operation: logical AND/OR

Summary

Groovy syntax

Scripting

Functional programming

- closures
- functors (map)
- monoids (reduce)

References

http://groovy-lang.org

http://grails.org