

# NPRG014 – 2020 plan

5th Oct - Groovy (FP, scripting) (VP)

12th Oct - Groovy (dynamism, meta-programming) (VP)

19th Oct - Groovy (DSL) (VP)

26th Oct - Concurrency (VP)

2nd Nov – Bytecode, Scala (TB)

9th Nov - Scala (TB)

16th Nov - Scala (TB)

23rd Nov - Scala (TB)

30th Nov - Prototype-based languages - IO (TB)

7 Dec - Prototype-based languages - JavaScript, TypeScript (TB)

# Criteria

A homework assignment will be given at each lecture

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

Submit **at least 8** out of 10 correctly implemented homeworks

# Repository

<https://github.com/d3scomp/NPRG014>

Clone and then checkout before each lecture

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

# Language dynamism, scripting and functional programming



**Václav Pech**

*NPRG014 2020/2021*

<http://www.vaclavpech.eu>

@vaclav\_pech

# Today's agenda

Groovy syntax and interoperability

Language dynamism

Scripting

Functional programming

# Groovy



A JVM programming language

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



### **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



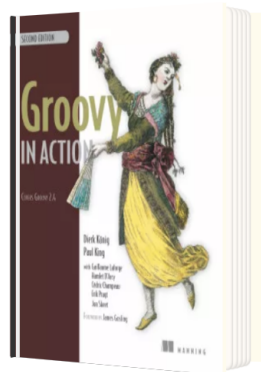
### **Domain-Specific Languages**

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



## The 7 usage patterns

- Super Glue
- Liquid Heart
- Keyhole Surgery
- Smart Configuration
- Unlimited Openness
- House-Elf Scripts
- Prototype



Examples in Groovy

**canoo**



# 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

```
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

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



# Groovy

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

# Groovy

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

# Groovy

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

# Groovy

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

# Groovy

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

# Groovy is Java

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

# Variables, constants, params

String a

def a – a re-assignable variable

final a – a constant value

# 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 =  $\{a, b \rightarrow a * b\}$

# 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  $\rightarrow$  a \* b}

def partial = multiply.curry(3)

def curried1 = {x  $\rightarrow$  {y  $\rightarrow$  multiply(x, y)}}

def curried2 = {x  $\rightarrow$  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`

# 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
- Grabbing libraries

# Scripting

Evaluate custom Groovy code

**At run-time!!!**

```
new GroovyShell().evaluate('println Hi!')
```

<http://groovyconsole.appspot.com/>



# Script customization

*CompilerConfiguration*

*CompilationCustomizer*

ImportCustomizer

ASTCustomizer

SecureASTCustomizer

# Functors

Dealing with wrapped data

$\text{map}: ([A], f: A \rightarrow B) \rightarrow [B]$

$\text{map}: (\text{Maybe}\langle A \rangle, f: A \rightarrow B) \rightarrow \text{Maybe}\langle B \rangle$

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:  $([A], f: (A, A) \rightarrow A) \rightarrow A$

# 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

transform

vs.

class CustData {orders, totalAmount}

a monoid



# Reduce vs. Fold

*m.reduce* {*v1*, *v2*  $\rightarrow$  *v1* + *v2*}

*m.foldLeft*(0) {*acc*, *v*  $\rightarrow$  *acc* + *v*}

# Composing functions

$f: A \rightarrow B$

$g: B \rightarrow C$

$f \gg g: A \rightarrow C$

# Composing functions

$f: A \rightarrow B$

$g: B \rightarrow C$

$f \gg g: A \rightarrow C$

```
def f = {String s → s.size()}
```

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

```
def h = f >> g
```

# Composing functions

$f: A \rightarrow B$

$g: B \rightarrow C$

$f \gg g: A \rightarrow C$

Not a monoid

# Endofunctors

$f: A \rightarrow A$

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

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

# Other monoids of functions

Elements:  $f: \text{String} \rightarrow \text{Boolean}$

# Other monoids of functions

Elements:  $f: \text{String} \rightarrow \text{Boolean}$

`id()` – returns *true/false*

Operation: logical AND/OR

# Summary

The joy of Ruby for Java programmers

[vaclav@vaclavpech.eu](mailto:vaclav@vaclavpech.eu)





# References

<http://groovy-lang.org>

<http://grails.org>

<http://groovyconsole.appspot.com/>

<http://www.manning.com/koenig2/>