Groovy

Agilný dynamický jazyk pre platformu Java

Róbert Novotný

(apríl 2013)

Skriptovacie jazyky

- cieľ klasické jazyky: **udržovateľnosť** / **výkonnosť**
  - nízkoúrovňové: C, C++: výkonnosť
  - obvykle objektové jazyky: Java, C#, ...: určené na megaprojekty
- existuje však kopa úloh, kde je dôležité ,,rýchlo niečo zbúchať", aby to fungovalo
  - udržovateľnosť ide bokom
  - výkon programu nie je dôležitá
  - rapid prototyping: nástrel softvéru, na ktorom sa odladí dizajn a utrasú používateľské požiadavky

História

- Dávno-pradávno: Unix napísaný v C, C napísané v Unixe
- Písať programčeky v C však nie je ktoviečo
- UNIX našťastie:
  - mnoho jednoúčelových nástrojov spájaných do kolóny
  - shell skripting
- Čo keby sme mali jazyk, ktorý niečo také zvláda?
- prelom 80./90 rokov: **Perl**

Niektoré piliere skriptovacích jazykov

Hlavne nech to ide!

- nie rýchlosť

Časté veci do knižnice!

- spracovanie textu
- práca so zoznamami
- ľahké IO
- XML

Účel svätí prostriedky

- procedurálne
- funkcionálne
- objektové

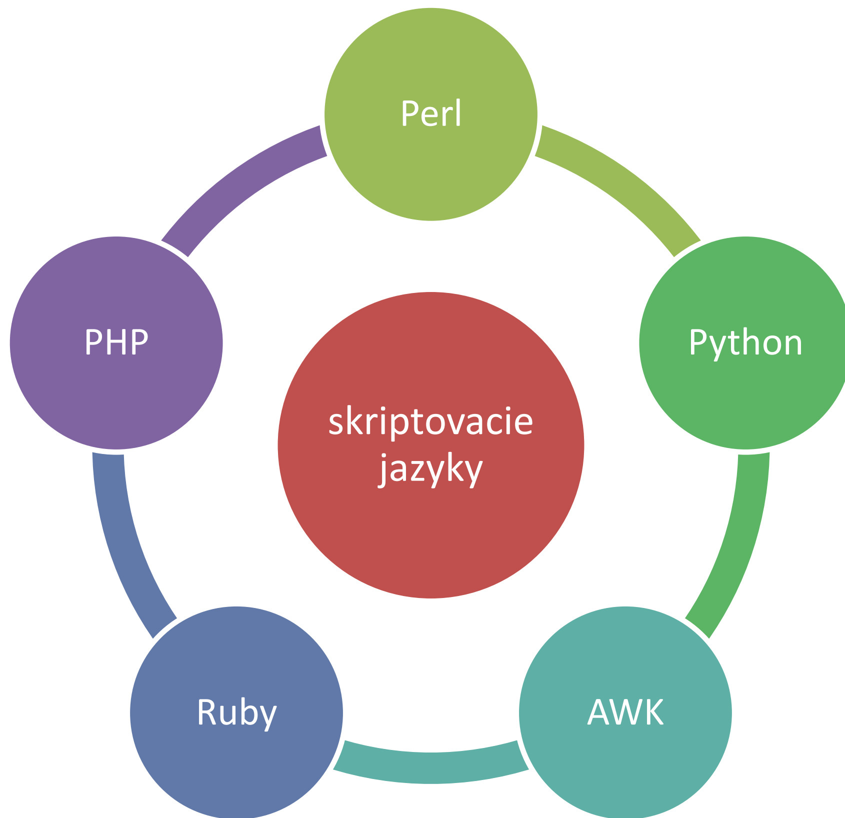
Kompilácia nie je nutná

- skript sa interpretuje dokola

Dátové typy zdržujú

- typy sa určia za behu
- alebo vôbec neexistujú

Expanzia skriptovacích jazykov



- AWK: spracovanie textových súborov
- Perl: praotec
- PHP: pôvodne webový, dnes univerzálny
  - pôvodne výhonok Perlu
- Python: objektovo-orientovaný, integrácia s C
  - fyzici ho milujú
- Ruby: ,,silnejší než Perl, viac OO než Python"

Java a skriptovacie jazyky

- Java je stabilne top jazyk na vývoj veľkých systémov
- lenže má nevýhody
  - ukecaná
  - treba virtuálny stroj
  - bez IDE sa takmer nič zmysluplné nedá napísať
- lenže má výhody
  - multiplatformnosť
  - tona knižníc
  - bublajúci kotol nových technológií
  - istota v zamestnanosti

Čo keby sme spojili výhody?

- multiplaformnosť + knižnice + jednoduchosť skriptovania
- riešenie už existuje
  - Jython = Python na Jave
  - JRuby = Ruby na Jave
  - Rhino = JavaScript na Jave

Groovy!

Groovy

- dynamicky typovaný OOP jazyk
- beží nad JVM
- základná idea: keďže poznáme Javu, odpichneme sa od nej a dodáme veci, ktoré poznáme z iných jazykov a chceme ich mať v Jave
- Groovy = Java++ ☺

Java


```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World");
  }
}
```

Groovy


```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World");
  }
}
```

Java vs Groovy: nájdi 2 rozdiely

groovy

- kde je rozdiel?

```
println "Hello World"
```

HelloWorld.groovy

```
c:\java\groovy\bin> groovy HelloWorld.groovy
```

skript bude interpretovaný!

groovyconsole.appspot.com

Groovy web console

```
1
2 println "Hello World"
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Actions ➣ | Execute script | New script | Publish script | View recent scripts

Subscribe to this site

Result | **Output** | **Stacktrace**

```
Hello World
```

Integrácia s Javou

- integrácia s Javou funguje magicky!
- ľubovoľnú triedu importneme a ideme
  - klasické balíčky sú automaticky importnuté (java.util, java.net...)
- ale i naopak: skompilovaný Groovy súbor je klasický .class
- vieme ho tak normálne používať v bežnej Jave

Dynamicky typovaný jazyk

- dátové typy nemusíme uvádzať

```
message = "Hello"
println message
```

- všimnime si
  - bodkočiarky nemusíme písať
  - dátový typ pre `message` sa určí automaticky
  - `println` je voľne poletujúca funkcia (sťa Pascal)
  - dokonca ani zátvorky netreba písať

Dynamicky typovaný jazyk

- dátový typ sa určí podľa priradzovanej hodnoty
- v situácii keď nepoznáme dátový typ, použijeme **def**

```
def message = "Ahojte!"
println message
```

```
String message = "Ahojte!"
println message
```

- v skriptoch môžeme **def** vynechať
- premenná sa stane globálnou

Kolekcie: skrátený zápis

- Java má svojský zápis pre inicializáciu kolekcií

```
List<String> mená = Arrays.asList(
  "John", "Graham", "Terry");
```

- Groovy zoznam:

```
mená = ["John", "Graham", "Terry"]
```

- Groovy mapa:

```
mapa = [meno: "Iron", priezvisko: "Maiden"]
```

kľúč sa automaticky prevedie na String

Kolekcie

- získanie hodnoty: C# / PHP style

```
println mená[0]
println mapa[meno]
```

  - ak chceme, môžeme klasicky **mená.get(0)**

- pridanie prvku

```
mená << "Ringo"
// alebo
novéMená = mená + "Ringo";
```

  - ak chceme, môžeme klasicky **`mená.add()`**

Preťažovanie operátorov

- objekt má metódy, na ktoré sa mapuje volanie operátorov
- `<<` sa mapuje na `leftShift()` na **java.util.Collection**
- `[]` sa mapuje na `getAt()`
- ...
- preťažovanie bolo z Javy zámerne vynechané (zlé skúsenosti z C++)

Triedy

groovy

- opäť jednoduchá syntax

```
class Pes {
   def rasa;
   def vek;
}
```

```
def pes = new Pes(rasa: "chrt", vek: 10)
println pes.rasa
println pes.vek
```

automaticky generované konštruktory

- inštančné premenné môžeme uviesť s typom alebo cez **def**
- ak neuvedieme modifikátor viditeľnosti (private...), automaticky sa vygenerujú gettre a settre

Prístup k premenným

groovy

- používame bodkovú notáciu, automaticky sa volajú gettre a settre
  - **pes.majiteľ.meno**
    - pes.majiteľ.meno = "Jozef";
  - **pes.getMajiteľ().getMeno()**
- existuje bezpečná navigácia
  - **pes?.majiteľ?.meno** = funguje aj v prípade, že je pes.getMajiteľ() == null
    - inak by sme dostali NullPointerException

Reťazce

- reťazce fungujú klasicky, ale môžeme v nich používať odkazy na premenné

```
class Pes {
  def rasa
  def vek
  String toString() {
    return "Rasa: ${rasa} – Vek: ${vek}";
  }
}
```

Interpolácia. Dosadia sa skutočné hodnoty.

Intervaly (ranges) a cykly


```
for(i in 0..2) {
  println "Hello"
}
```

- interne reprezentované ako zoznamy

```
0..<3
```

sprava otvorený

```
0..<3.contains(3) == false
```

interval je zoznam

Funkcie

groovy

- funkcie môžeme pchať kam len chceme, nemusia byť v metódach
- ak nevieme, aký typ vraciame, píšeme def
- parametre môžu mať implicitné hodnoty

```
def vypíš(hodnota, opakovaní = 3) {
  for(i in 0..<opakovaní){
    println hodnota
  }
}
vypíš("Budem si robiť DÚ")
```

Kamenitá cesta k funkcionálku

- **Úloha!** Z daného poľa vráťte len tie prvky, ktoré spĺňajú podmienku

  napríklad žiakov, ktorých mená sa začínajú na "Z"

- klasické riešenie:
  - vytvorme nové pole
  - prechádzame pôvodné pole **for**-om
  - if (prvok spĺňa podmienku) potom hoď do výsledného poľa
  - vráť výsledné pole

Kamenitá cesta k funkcionálku

- čo ak chceme raz študentov "Z"-ovcov?
- potom zase študentov, ktorých mená začínajú na "A"
- potom študentov, ktorí majú aspoň 1 predmet?
- potom študentov, ktorí....

- zbesilo kopírujeme cyklus, mení sa len podmienka
- úloha typu: zober **dáta** a **niečo** s nimi **sprav**

zoznam študentov

iteruj

filtruj

Kamenitá cesta k funkcionálku

- podmienka "začína na Z" je vlastne pravdivostná funkcia
  - berie študenta, vracia true/false
- všetky úlohy sú potom o:
  - zober **dáta**, zober **funkciu**
  - na každý prvok **dát** aplikuj **funkciu**
- ak sú dáta **objektami**, prečo aj funkcia nemôže byť **objektom**?

Funkcia môže mať ľubovoľný kód, nielen pravdivostný
Kód sa stáva objektom!

Funkcionálna mentalita

- našu úlohu vyrieši funkcionálnik takto:
  - funkcionálnik vytvorí inštanciu funkcie....
    - lambda výraz a.k.a. closure
  - ...ktorá má jeden parameter a vracia parameter
- metóda filtrujúca študentov potom dostane
  - 1 zoznam (dáta)
  - 1 closure (funkcia)
- môžeme tak zuniverzálniť problém

Funkcionálna mentalita


```
def hľadajŠtudentov(List<Student> dáta, Closure funkcia)
{
  def filtrovaniStudenti = []
  for(Student s : dáta) {
    if(funkcia.call(s)) {
      filtrovaniStudenti << s
    }
  }
}
```

```
def podmienka = { Student student ->
  return student.priezvisko.startsWith("Z")
}
```

```
def študenti = ...
println hľadajŠtudentov(študenti, podmienka)
```

Closures sú na každom kroku

groovy

- **Úloha!** Vypíšte trikrát Ahoj svet!

```
for(int i = 0; i < 3; i++) {
  System.out.println("Ahoj svet!");
}
```

Java mentalita

- Trikrát vykonaj closure, ktorý nemá parameter a vracia nič!

```
3.times {
   println "Ahoj svet!"
}
```

Groovy mentalita

Closures sú na každom kroku

groovy

- Úloha! Vypíšte trikrát Ahoj svet!

```
3.times {
   println "Ahoj svet!"
}
```

Groovy mentalita

- 3 je konštanta
- čísla v Groovy sú objekty (v Jave: primitívy) s metódami
- `times()` je metóda na čísle
- ako parameter berie closure, ktorý sa má vykonať

Closures sú objekty


```
def druháMocnina = {
  Integer číslo -> return číslo * číslo
}
```

```
println druháMocnina(2)
```

- naľavo od šípky: deklarácia argumentov funkcie
- napravo od šípky: kód sťa v normálnej metóde

```
println druháMocnina.call(2)
```

closure je objekt typu **groovy.lang.Closure**

Closures sú objekty

groovy

```
def druháMocnina = {
  Integer číslo -> return číslo * číslo
}
```

```
def druháMocnina = {
  číslo -> return číslo * číslo
}
```

dátový typ netreba

```
def druháMocnina = {
  return it * it
}
```

closure má automatický parameter **it**

```
def druháMocnina = { it * it }
```

návratovou hodnotou je výsledok posledného vykonaného výrazu

Iterovanie cez kolekcie: each


```
def mená = ["John", "Graham", "Terry"]
mená.each {
  println it
}
```

- **each** berie parameter typu closure
- **it** automaticky dostupná premenná v closure, tu obsahuje aktuálny prvok

Hľadanie prvého prvku spĺňajúceho podmienku: **find**

groovy

```
def mená = ["John", "Graham", "Terry"]
def gMeno = mená.find {
  it.startsWith("G")
}
```

- **find** berie parameter typu closure
- closure určuje booleovskú podmienku pre hľadaný prvok
- **it** automaticky dostupná premenná v closure, tu obsahuje aktuálne skúmaný prvok

Hľadanie všetkých prvkov spĺňajúcich podmienku: **findAll**

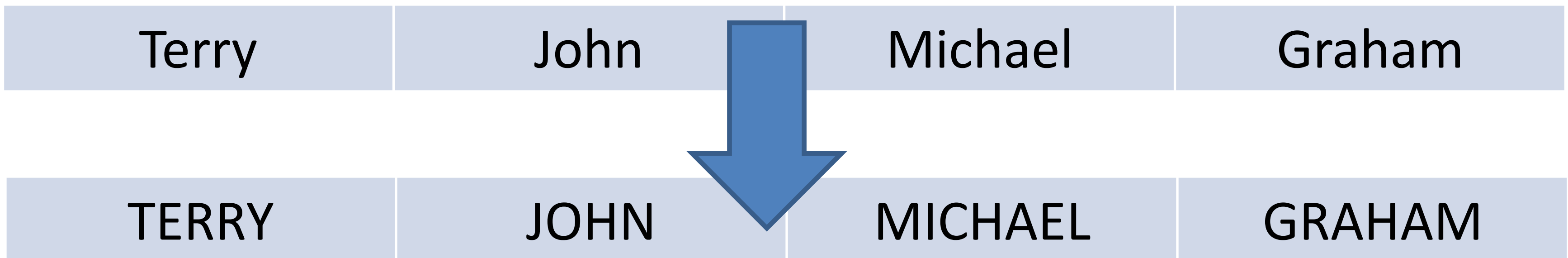

```
def mená = ["John", "Graham", "Terry"]
def gMená = mená.findAll {
  it.startsWith("G")
}
```

- **findAll** berie parameter typu closure
- **it** automaticky dostupná premenná v closure, tu obsahuje aktuálne skúmaný prvok

Transformácie vstupných prvkov: **collect**

groovy

```
veľkéMená = mená.collect{
  it.toUpperCase()
}
```



- vráti nový zoznam, kde každý prvok prejde nejakou transformáciou
- transformácia je daná pomocou closure

Transformácie vstupných prvkov: **collect**

- alternatívny zápis

```
veľkéMená = mená*.toUpperCase()
```

- samozrejme, môžeme mapovať prvok na úplne iný typ
- vhodné pre úlohy, kde mapujeme prvok na číslo

Komplexný príklad: indický vynálezca šachu

- odmena za vynález šachu? zrno na políčkach šachovnice.
- na prvom políčku 1 zrnko, na druhom 2, na každom dvojnásobok predošlého políčka

$$T_{64} = 1{+}2{+}4{+}\cdots{+}9,223,372,036,854,775,808 = 18,446,744,073,709,551,615$$

$$T_{64} = 2^0 + 2^1 + 2^2 + \cdots + 2^{63}$$

```
(0..<64).collect { 2 ** it }.sum()
```

Ako hrať Swing jednoduchšie


```
final JButton button = new JButton("Push me!");
button.addActionListener(new ActionListener(){
  public void actionPerformed(ActionEvent event){
    System.out.println(button.getText());
  }
});
```

- po closures najviac túžia vývojári Swingu
- Java syntax je totiž dosť hrôzostrašná
  - anonymné vnútorné triedy...

Ako hrať Swing jednoduchšie


```
def button = new JButton("Push me!")
button.actionPerformed = {
  println button.text
}
```

- listenery sú nič iné než kusy kódu, ktoré sa majú vykonať po vyvolaní udalosti
- v Jave: vo verzii 8 (jeseň 2013)

Príklad na SQL


```
import groovy.sql.Sql
sql = Sql.newInstance(
   "jdbc:mysql:localhost/test",
   "username", "password",
   "com.mysql.jdbc.Driver")

sql.eachRow("select * from tableName",
         { println it.id + " -- ${it.firstName} --"}
);
```

- **eachRow** berie dva parametre: dopyt a closure, ktorý sa zavolá pre každý riadok

Práca so súbormi je malina


```
new File("studenti.txt").eachLine {
  println it
}
```

- otvorenie a zatvorenie súboru sa deje automaticky

```
súbor = new File("studenti.txt");
súbor << "Jozef Mak"
súbor << "Ringo Starr"
```

- stále máme možnosť používať klasický prístup

```
out = new File("studenti.txt").newPrintWriter()
out.println("Jozef Mak")
```

Duck typing


```
class Kačka {
    plávaj() { println "Čľap čľap" }
}

class Žralok {
    plávaj() { println "Čľap ham" }
}

vodnéŽivočíchy = [new Žralok(), new Kačka()]
vodnéŽivočíchy.each {
    it.plávaj()
}
```

Ak to pláva ako kačka a lieta ako kačka, je to kačka.

V Jave by Kačka i Žralok museli implementovať interfejs Plávajúci s metódou plávaj().

Expandos: totálne dynamické triedy


```
pes = new Expando()
println pes.meno //null

pes.meno = "Rex"
println pes.meno //Rex

pes.stekaj() //null
pes.stekaj = { počet ->
   return "haf!" * počet
}
println pes.stekaj
```

náš pes na začiatku nevie nič!

naučíme ho vlastné meno

naučíme ho štekať

Expandos umožňujú za behu pridávať inštančné premenné a metódy.

Regulárne výrazy

- Umožňujú filtrovať text, vyhľadávať...
  - veľký hit v Perle a Awku
- špeciálna syntax: operátor =~

```
zhoda = "Ringo Starr" =~ /^R/
```

začína sa reťazec na R?

- vytvára objekt **java.text.Matcher**
- ak treba vyhodnotiť podmienku, vracia true v prípade, že je regulárny výraz splnený

Regulárne výrazy


```
mená = ["Ringo Starr", "Terry Jones", "Terry Gilliam"]
menáTG = mená.findAll {
    it =~ /T(.+) G/
}
```

Má meno iniciály T G?

```
dáta = [2009, 2010, 2OLL, 2011]
println dáta.findAll { it ==~ /\d+/}
```

==~

Pozostáva vstup z cifier?

Regulárne výrazy

- operátor =~ hľadá zhodu podreťazca
- operátor ==~ hľadá celý reťazec

```
mená = ["Ringo Starr", "Terry", "Terry Gilliam"]
terryovci = mená.findAll {
    it =~ /Terry/
}
```

"Terry", "Terry Gilliam"

```
mená = ["Ringo Starr", "Terry", "Terry Gilliam"]
terryovci = mená.findAll {
    it ==~ /Terry/
}
```

"Terry"

Regulárne výrazy: skupiny


```
dáta = "meno=Peter priezvisko=Malý mesto=Košice"

matcher = (dáta =~ /\S+=(\S+)/)
matcher.each { println it[1] }
```

slovo, znak =, *slovo*

- matcher sa dá iterovať
- hľadá všetky zhody (v príklade: 3 zhody)
- každá zhoda je pole
  - 0. prvok: celý text zhody
  - 1. prvok: obsah prvej skupiny, 2. prvok = 2. skupina...

- trieda **XMLSlurper** vráti XML ako objekt, na ktorom možno volať inštančné premenné

```
def rss = new XmlSlurper().parse(
  "http://web.ics.upjs.sk/webcalendar/rss.php")
rss.channel.item.each{ println it.title }
```

názvy akcií na ÚINF PF UPJŠ

```
rss.channel.item[0..<3].each{ println it.title }
```

tri najbližšie akcie

XML

- **MarkupBuilder** umožňuje budovať XML
  - návrhový vzor Builder je všadeprítomný
  - umožňuje ľahko definovať doménovo špecifické jazyky (DSL)

```
import groovy.xml.*;
new MarkupBuilder().psi
{
  pes( id:1 ) {
    meno("Rex")
    vek(5)
  }
}
```

```
<psi>
  <pes id='1'>
    <meno>Rex</meno>
    <vek>Mak</vek>
  </pes>
</psi>
```

SwingBuilder: rýchle budovanie UI

import groovy.swing.SwingBuilder
import java.awt.BorderLayout as BL

```
pocet = 0
new SwingBuilder().edt {
  frame(title:'Frame', size:[300,300], show: true) {
    borderLayout()
    textlabel = label(text:"Kliknite na tlačidlo!",
                      constraints: BL.NORTH)
    button(text:'Klikni!',
           actionPerformed: {
             count++
             textlabel.text = "Počet klikov: ${pocet}"
           },
           constraints:BL.SOUTH)
  }
}
```

Zhrnutie

- Groovy je Java++
- opravuje množstvo kritík smerovaných na jazyk
  - prečo primitívy a objekty?
  - prečo porovnávame cez equals()
  - prečo odchytávať mnoho výnimiek?
  - prečo je I/O zložité?
  - prečo nemáme delegátov z C#?
  - prečo na HelloWorld potrebujem triedu a päť riadkov?
  - prečo nemáme preťaženie operátorov?

Zhrnutie

- Groovy poskytuje možnosť používať plný repertoár Javy
- a dodáva množstvo syntaktického cukru
- výborné na rýchle prototypovanie aplikácií
- rýchly vývoj má daň
  - udržiavateľnosť – množstvo vecí sa deje "automaticky", kód sa rýchlo píše, ale zložito číta
  - môžeme prísť o výhody ako refactoring...

Literatúra

- http://www.root.cz/serialy/groovy-v-prikladech/ - **český seriál**
- **Tutoriál Fluently Groovy (IBM):** http://www.ibm.com/developerworks/edu/j-dw-java-jgroovy-i.html
- **Groovy in Action** (Manning, 2007)
- **Groovy Applet:** http://metawidget.sourceforge.net/live-demo/demo.html