Groovy

Michał Lenart

15.XII.2008

Plan prezentacji

Czym jest Groovy

Definicja z Oxford Advanced Learner's Dictionary

groovy *adjective* (old-fashioned, informal) fashionable, attractive and interesting

Groovy to skryptowy, obiektowy język programowania ze składnią wzorowaną na Javie, inspirowany językami takimi jak Ruby, Python czy Smalltalk. Posiada wiele funkcjonalności ułatwiających życie programisty:

- Domknięcia
- Przeciążanie operatorów
- Operacje na kolekcjach (takie, jak np. w Pythonie)
- Obsługa wyrażeń regularnych
- Dynamiczne typowanie
- i wiele innych...

Dlaczego warto używać Groovy

- Dla miłośników Javy:
  - Podobna składnia
  - Biblioteki Groovy i Javy mogą ze sobą łatwo współpracować
- Wygodny w użyciu - bardziej czytelny i krótszy kod
- Grails - zrąb do tworzenia aplikacji WWW

Groovy - język skryptowy, ale z możliwością kompilacji

Są dwa sposoby tworzenia klas Groovy:

- Dynamicznie, za pomocą groovysh lub groovyConsole
- Kompilacja do pliku .class za pomocą groovyc

Hello world

- Najprostsza wersja:

```
println "Hello world"
```

- Można też w ten sposób:

```
class Hello {
    def sayHello() {
        println "Hello world"
    }

    static void main(args) {
        new Hello().sayHello()
    }
}
```

Podstawowe elementy składni

```
// deklaracja zmiennych
def x
def napis = "asfasdf"

// instrukcja warunkowa
if (x == 123)
    println x // wypisanie zmiennej na stdout
else
    println "różne od 123"

/*
 * pętla i blok kodu
 */
while(!true) {
    println "to się nigdy nie powinno wyświetlić"
}
```

Dynamiczne typowanie

- Typ obiektu jest sprawdzany w czasie wykonania (a nie kompilacji)
- Przykład 1 - zupełnie poprawny

```
def x = "asdfa"
x = 1234
```

- Przykład 2 - uruchomi się, ale wyrzuci wyjątek:

```
def x = "asdfasdf"
x.zrobCosZeSoba()
```

- Można też ustalić pożądany typ w momencie deklaracji. Przykład 3 - uruchomi się, ale wyrzuci wyjątek:

```
int x = 1234
x = new Object()
```

Obiektowość

- **Wszystko jest obiektem** - brak typów prostych
- Wszystko dziedziczy po Object
- Generalnie wszystko podobnie do Javy

```
interface X {
  def x()
}
interface Y {
  def y()
}
abstract class A {
  def a(){ println 1 } // definicja metody
  abstract b()         // metoda abstrakcyjna
}
class B extends A implements X, Y {
  def x(){ println 2 }
  def y(){ println 3 }
  def b(){ println 4 } // z klasy A
}
```

Domknięcia

- Domknięcie - anonimowy fragment kodu, który może zostać zapamiętany i wykonany w dowolnym momencie, a ponadto:
  - pobierać parametry
  - dawać wynik
  - korzystać ze zmiennych zadeklarowanych na zewnątrz
- Domknięcia pełnią podobną funkcję jak bloki w Smalltalku.
- Przykład (bardzo elementarny):

```
def closure = { println "Hello!" } // tylko przypisanie
closure() // tutaj wypisuje "Hello!" na stdout
```

- Przykład - z pobieraniem parametrów:

```
def iloczyn = { x, y -> return x * y }
println iloczyn(6, 7) // wypisze 42
```

Domknięcia

- Pytanie - co zostanie wypisane na ekran:

```
class X {
def getX() {
        def x = 1234
        def res = { x }
        x = "qwerty"
        return res
}

    static void main(args) {
        def c = new X().getX()
        println c()
    }
}
```

- Odpowiedź - oczywiście "qwerty"

Domknięcia

- Przykład z życia - zastąpienie anonimowych klas wewnętrznych (anonymous inner classes) z Javy domknięciami.
- Kod w Javie (brzydki):

```
button.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent event){
        zrobCosPozytecznego();
    }
});
```

- Ten sam kod w Groovy:

```
button.addActionListener { zrobCosPozytecznego() }
```

- Generalnie dzięki temu mamy bardziej przejrzysty kod

Domknięcia

UWAGA: W języku Groovy nie można stosować bloków kodu w dowolnym miejscu (jak w Javie)!

- To jest poprawne - w nawiasach ""jest blok kodu:

```
if (true) {
    obiekt.zrobCos()
}
```

- Natomiast to NIE zadziała poprawnie:

```
def x = 1
{
    x = 2 // Domknięcie czy blok kodu??
}
```

Tak nie należy pisać!

Przeciążanie operatorów

- Aby przeciążyć operator trzeba zdefiniować odpowiednią metodę
- Przykłady (nie wszystkie):

Operator	**Metoda**
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.multiply(b)
a[b]	a.getAt(b)
a[b] = c	a.putAt(b, c)
switch(a) { case(b) : }	b.isCase(a)

Operator '=='

- NIE jest to to samo, co '==' w Javie
- Aby sprawdzić, czy a i b to ten sam obiekt, trzeba zrobić tak:

```
boolean areTheSame = a.is(b)
```

- Natomiast '==' znaczy to samo, co equals

```
boolean areEqual = a == b
areEqual = a.equals(b) // to to samo
```

Operator isCase(object)

- Jak to działa - przykład:

```
switch (x) {
    case a: zrobA(); break // jeśli a.isCase(x)
    case b: zrobB(); break // jeśli b.isCase(x)
    default: nicNieRob()
}
```

- Dzięki temu switch potrafi dużo więcej, niż w Javie...

```
switch (x) {
   case 0 : println "zero" ; break
   case 0..9 : println "cyfra" ; break
   case Float : println "float" ; break
   case {it % 3 == 0}: println "parzyste"; break
   case ~/../ : println "cośtam" ; break
   default : assert false ; break
}
```

Operator "Elvis" ?:

- Szczególny przypadek operatora "a ? b : c"

```
// standardowy operator znany z C/C++/Java
def gender = user.male ? "male" : "female"

// jeśli user.name == null to dostaniemy "Anonymous"
def displayName = user.name ?: "Anonymous"
```

Operator bezpiecznej nawigacji ?.

- Można pisać tak, żeby uniknąć NullPointerException:

```
if (user != null && user.address != null)
    street = user.address.street
else
    street = null
```

- W Groovy można tak:

```
street = user?.address?.street
```

Dużo ładniej, a robi dokładnie to samo :)

Napisy

- Standardowy napis - taki jak w Javie:

```
def str = 'Jakiś napis'
assert str instanceof java.lang.String
```

- GString - dodatkowo można dodawać parametry (tak jak w bashu czy PHP):

```
def x = "asdf"
def str = "Jakiś napis $x" // piszemy " zamiast '
assert str == "Jakiś napis asdf"
```

  Wartości parametrów wyliczane leniwie - dopiero przy pytaniu o wartość!

- Napisy w Groovy mają dodatkowe możliwości, np:

```
str.eachLine { line -> zrobCosZ(line) }
```

Listy

- Tworzenie listy i dodawanie elementów:

```
def list = [1, 2, 3, 4, 3]
list << "asdf" // dodanie "asdf" na koniec listy
list -= 3
// list == [1, 2, 4, "asdf"]
```

- Domyślnie używane ArrayList. Ale można też tak:

```
def list = new LinkedList()
list << "element1"
// ...
```

- Uwaga:

```
list += 1
list -= 2
```

Przy użyciu '+=' i '-=' za każdym razem tworzona jest nowa lista

Zakresy

- Zakres jest to lista kolejnych wartości. Przykład:

```
def range = 1..100
assert range.size() == 100
assert range.from == 1 && range.to == 100
assert range.contains(10)
assert range instanceof List
assert !range.contains(101)
```

- Implementacja w czasie stałym (klasa Range) - pamięta tylko "from"i "to"
- Można też tworzyć zakresy dla dowolnych obiektów - nie tylko liczb

Mapy

- Przykład użycia mapy:

```
def map = [key:"value", (1):"jeden", (x):"iks"]

map.key2 = value2
map["asdf"] = 13
map[2] = "dwa"
```

- Tworzenie pustej mapy:

```
def map = [:]

// oczywiście można też tak:
def map = new LinkedHashMap()
```

Ulepszona pętla for

- W Groovy nie używamy pętli for tak jak w Javie:

```
for (int i: collection)
    System.out.println(i)
```

- Jest za to taka konstrukcja:

```
for (elem in collection)
    println elem
```

- Ponieważ Range dziedziczy po List, ten kod:

```
for (i in 1..100)
    println i
```

wypisze wszystkie liczby naturalne od 1 do 100

Domknięcia i kolekcje

- Dzięki domknięciom wiele operacji na kolekcjach jest dużo wygodniejszych:

```
assert col instanceof Collection
col.each { x -> zrobCos(x) }

// sortuj w odwrotnej kolejności
reverse = col1.sort { x, y -> y.compareTo(x) }

// czy chociaż jeden/wszystkie elementy spełniają warunek
col.any { x -> warunek(x) }
col.every { x -> warunek(x) }
```

- Przykład na mapie:

```
def map = [a:1, b:2, c:"c"]
def x = map.find { key, value -> key == value }
assert x.value == "c"
```

Operator *.

- Przykład użycia operatora:

```
def list = [1, 2, 3]
list2 = list*.multiply(2)
assert list2 == [2, 4, 6]
```

- Wersja równoważna:

```
list3 = list.collect { x -> 2*x }
```

GroovyBeans

- Aby klasy tworzone przez Groovy mogły być stosowane w Javowych zrębach aplikacji (np. JSP, EJB, JPA) powinny spełniać specyfikację JavaBeans
- JavaBeans to klasy Javy spełniające dodatkowe konwencje:
  - posiadanie publicznego konstruktora bez parametrów
  - atrybuty dostępne są przez akcesory (get, set)
  - klasa powinna być serializowana

GroovyBeans

Przykład JavaBean:

```
public class MyBean implements java.io.Serializable {
    private String myprop;
    public String getMyprop(){
        return myprop;
    }
    public void setMyprop(String value){
        myprop = value;
    }
}
```

Równoważna klasa Groovy:

```
class MyBean implements Serializable {
    String myprop
}
```

GroovyBeans

Przykład - kolejny Bean:

```
class MrBean implements Serializable {
    String firstname, lastname
    String getName(){
        return "$firstname $lastname"
    }
}
```

oraz przykład jego użycia:

```
def bean = new MrBean(firstname: 'Rowan')
bean.lastname = 'Atkinson'
assert 'Rowan Atkinson' == bean.name
```

BuilderSupport

- Klasa BuilderSupport służy do wygodnego tworzenia struktur drzewiastych. Przydatne do takich zastosowań jak tworzenie dokumentu HTML, XML lub GUI (np. w Swingu)
- W wielu językach są narzędzia, które to robią poprzez metody typu:
  - addChild(node)
  - addAttribute(attr)
  - setParent(node)

  Nie jest to ani wygodne, ani czytelne
- Za pomocą podklas BuilderSupport tego typu struktury można tworzyć *bezpośrednio w kodzie*

Przykładowy XML

Chcemy wygenerować taki dokument XML:

```
<?xml version="1.0"?>
<hotel>
  <pokoj nr="1">
    <gosc>Jan Kowalski</gosc>
  </pokoj>
  <pokoj nr="2">
    <karaluch gatunek="Blatella germanica"/>
    <karaluch gatunek="Blatella germanica"/>
    <karaluch gatunek="Blatella germanica"/>
  </pokoj>
</hotel>
```

Generowanie XML-a w Javie

```
Element hotel = doc.createElement("hotel")
Element pok1 = doc.createElement("pokoj")
Element pok2 = doc.createElement("pokoj")

doc.appendChild(hotel)
hotel.appendChild(pok1)
hotel.appendChild(pok2)

pok1.setAttribute("nr", "1")
pok2.setAttribute("nr", "2")

Element gosc = doc.createElement("gosc")
pok1.appendChild(gosc)
gosc.setTextContent("Jan Kowalski")

for (int i = 0; i < 3; i++) {
    Element kar = doc.createElement("karaluch")
    kar.setAttribute("gatunek", "Blatella germanica")
    pok2.appendChild(kar)
}
```

Groovy - MarkupBuilder

- Wersja z użyciem DOM API była brzydka. Jak to zrobić ładniej?
- W groovy używamy klasy MarkupBuilder:

```
def writer = new StringWriter()
def xml = new MarkupBuilder(writer)

xml.hotel {
    pokoj(nr:"1") {
        gosc "Jan Kowalski"
    }
    pokoj(nr:"2") {
        for (i in 1..3)
            karaluch(gatunek:"Blatella germanica") {}
    }
}
// wypisz xml
println writer.toString()
```

MarkupBuilder i Groovlets

W Groovy można pisać (tak jak w Javie) Servlety. Można jest nazwać "Groovlety". Dzięki MarkupBuilder można czytelnie generować kod html:

```
import java.util.Date
import groovy.xml.MarkupBuilder

session = session ?: request.getSession(true);
session.counter = session.counter ?: 1

html.html { // html == new MarkupBuilder(System.out)
  head {
      title("Groovy Servlet")
  }
  body {
    p("Hello world!")
  }
}
session.counter = session.counter + 1
```

Budowanie GUI - SwingBuilder

- Swing - standardowa biblioteka do tworzenia GUI w Javie
- Posiada mnóstwo gotowych komponentów do tworzenia:
  - okien
  - paneli
  - przycisków
  - tabel
  - i wielu innych...

SwingBuilder

Prosty skrypt używający GUI:

```
swing = new SwingBuilder()
frame = swing.frame(title:'Demo') {
    menuBar {
        menu('File') {
            menuItem 'New'
            menuItem 'Open'
        }
    }
    panel {
        label 'Label 1'
        slider()
        comboBox(items:['one','two','three'])
    }
}
```

SwingBuilder

Przykład obsługi akcji użytkownika - naciśnięcia przycisku:

```
swing = new SwingBuilder()
frame = swing.frame(title:'Printer')
swing.widget(frame) {
    panel {
        textField(columns:10)
        button(text:'Print', actionPerformed: {
            println frame.title
        })
    }
}
```

SwingBuilder

To samo, ale zapisane w inny sposób:

```
import groovy.swing.SwingBuilder

swing = new SwingBuilder()
button = swing.button('Print')
frame = swing.frame(title:'Printer') {
    panel {
        textField(columns:10)
        widget(button)
    }
}

button.actionPerformed = {
    println frame.title
}
frame.pack()
frame.show()
```

SwingBuilder

Jeszcze jeden przykład - tabela:

```
data = [
    [nick:'MrG', full:'Guillaume Laforge' ],
    [nick:'jez', full:'Jeremy Rayner' ],
    [nick:'fraz', full:'Franck Rasolo' ],
]
swing = new SwingBuilder()
frame = swing.frame(title:'Table Demo') {
    scrollPane {
        table() {
            tableModel(list:data) {
                propertyColumn(
                  header:'Nickname', propertyName:'nick'
                )
                propertyColumn(
                  header:'Full Name', propertyName:'full'
                )
        }
    }
}
```

Tworzenie własnego Budowniczego

- Istnieje możliwość tworzenia własnego Budowniczego
- W tym celu trzeba stworzyć podklasę BuilderSupport implementującą następujące metody:
  - createNode(name, attributes, value)
  - setParent(parent, child)
  - nodeCompleted(parent, node) - opcjonalnie
- Wówczas przykładowo foo('bar') powoduje "automagiczne"wywołanie createNode('foo', 'bar')

Tworzenie własnego Budowniczego

Przykładowo taki fragment kodu:

```
builder = new MyBuilder()
builder.foo() {
bar(a:1)
}
```

w rzeczywistości działa tak:

```
builder = new MyBuilder()
foo = builder.createNode('foo')
// setParent() w korzeniu nie ma sensu
    bar = builder.createNode('bar',[a:1])
    builder.setParent(foo, bar)
        // brak domknięcia dla 'bar'
    builder.nodeCompleted(foo, bar)
builder.nodeCompleted(null, foo)
```

Jak Groovy tworzy klasy Javy

- Problem - "duck typing- jedyne co na pewno wiemy o zmiennej **x**, to to, że jest klasy Object:

```
x.wykonajJakasMetode()
```

  W Javie dostaniemy NoSuchMethodException
- Tak naprawdę skompiluje się to do czegoś w rodzaju:

```
getMetaClass()
   .invokeMethod(this, "wykonajJakasMetode", [])
```

- Tak więc "obiekt.abc()"wcale nie wywołuje metody Object::abc(). Faktyczne wywołanie może być zrealizowane np. za pomocą mechanizmu refleksji
- Implementując interfejs GroovyInterceptible można przeciążyć operator ".nazwaMetody"

Podsumowanie

- Łatwa nauka - zwłaszcza dla programistów Javy
- Domknięcia, przeciążanie operatorów, dodatkowe operacje na kolekcjach i napisach zwiększają czytelność kodu
- Wygodne tworzenie struktur hierarchicznych dzięki podklasom BuilderSupport
- Stosowany m.in. do programowania serwisów WWW (dzięki zrębowi Grails)
- Istnieją wtyczki do Eclipse i NetBeans wspierające Groovy.

Minus:

- W porównaniu z Javą niestety gorsza wydajność - nie do uniknięcia.

Bibliografia

 Dierk Koenig with Andrew Glover, Paul King, Guillaume Laforge and Jon Skeet *Groovy in action*. January, 2007.

 Oficjalna strona projektu Groovy http://groovy.codehaus.org