

Ułatwienia programowania obiektowego w języku Groovy

© Krzysztof Barteczko, PJWSTK 2012-2017

Klasy w Groovy

Z programów w języku Groovy mamy dostęp do klas napisanych w Javie i odwrotnie.

Klasy w Groovy są jak w Javie, z pewnymi drobnymi różnicami i wieloma dodatkowymi właściwościami.

Metody i konstruktory są domyślnie publiczne.

Pola:

1. Jeśli zmienna jest deklarowana ze specyfikatorem dostępu (public, private lub protected) to oznacza pole.
2. Bez specyfikatora dostępu - oznacza właściwość (property) (zob. dalej).

Konstruktor:

wywołanie z nazwanymi argumentami (jeśli konstruktor niezdefiniowany),
domniemane wołanie przy przypisaniu listy (dla zdefiniowanych konstruktorów), przy przypisaniu map (bez zdefiniowanego).
Zob. dalej

Skrypty to też klasy

Plik nie zawierający klas == skrypt.

Generowana jest klasa o tej samej nazwie co plik. Zawartość pliku (kod) staje się ciałem metody run, dodatkowo jest tworzona statyczna metoda main.

Script A

```
println "I'm script A"  
args.each { print it + ' ' }
```

Script B

```
println 'Script B calling script A'  
ScriptA.main('a b c')
```

Output:

```
Script B calling script A  
I'm script A  
a b c
```

Inne przypadki

Pojedyncza klasa w pliku - dokładnie jak w Javie

Wiele klas w pliku - nie ma restrykcji co do nazw, ale gdzieś musi być metoda main.

W pliku można łączyć definicję klas i kod skryptu. Kod skryptu staje się wtedy klasą główną.

```
class Student implements Comparable {  
    def fname, lname  
    Student(name) { (fname, lname) = name.tokenize() }  
    public int compareTo(other) {  
        fname.compareTo(other.fname)  
    }  
    public String toString() { "$fname $lname" }  
}
```

Nazwa pliku:

ProblemWithSet

a nie "Student"

```
def list = [ new Student('John Dog'), new Student('Steve Horse'), new  
Student('John Cat') ]  
println list  
println(new TreeSet(list))
```

Output:

```
[John Dog, Steve Horse, John Cat]  
[John Dog, Steve Horse]
```

Gdzie jest John Cat?



Konstruktory

Gdy jest zdefiniowany - może być wywołany na dwa sposoby:

```
public class Person {  
    String fname, lname, info  
  
    Person(fn, ln, inf) {  
        fname = fn; lname = ln; info = inf;  
    }  
  
    public String toString() {  
        def list = []  
        if (fname) list << fname  
        if (lname) list << lname  
        if (info) list << info  
        return list.join(' ')  
    }  
  
    public static void main(String ... args) {  
        def plist = []  
        plist << new Person('Albert', 'Einstein', 'was here')  
        Person p1 = [ 'Albert', 'Einstein', 'scientist']  
        plist << p1  
        plist.each { println it }  
    }  
}
```

Pozycyjne arg



Domniemane wywołanie
przy przypisaniu listy arg



Output:

```
Albert Einstein was here  
Albert Einstein scientist
```

Gdy brak definicji konstruktora ...

```
public class Person {  
    String fname, lname, info  
  
    public String toString() {  
        def list = []  
        if (fname) list << fname  
        if (lname) list << lname  
        if (info) list << info  
        return list.join(' ')  
    }  
  
    public static void main(String ... args) {  
        def plist = []  
        plist << new Person()  
        plist << new Person(fname: 'Albert')  
        plist << new Person(lname: 'Einstein', info: 'great scientist')  
        Person p1 = [ fname: 'Albert', lname: 'Einstein']  
        plist << p1  
        plist.each { println it }  
    }  
}
```

Nazwane parametry



[Zob w dokumentacji
konstruktory](#)

Output:

```
Albert  
Einstein great scientist  
Albert Einstein
```

Właściwości

Model komponentowy
JavaBeans.

Konwencje nazewnnicze:

private T prop;

public T getProp() // getter

public void setProp(T) //setter

Groovy

```
public class GrooBean {  
    def text, counter  
    String toString() {  
        "$text $counter"  
    }  
}
```

[Zobacz materiał o
JavaBeans \(w Javie\)](#)

Java

```
public class JavaBean {  
    private String text;  
    private int counter;  
  
    public JavaBean(String text, int counter) {  
        this.text = text;  
        this.counter = counter;  
    }  
  
    public String getText() {  
        return text;  
    }  
  
    public void setText(String text) {  
        this.text = text;  
    }  
  
    public int getCounter() {  
        return counter;  
    }  
  
    public void setCounter(int counter) {  
        this.counter = counter;  
    }  
  
    public String toString() {  
        return text + " " + counter;  
    }  
}
```

Dostęp do właściwości

Java -> Java

```
public static void main(String[] args) {  
    JavaBean jb = new JavaBean("Text", 1);  
    System.out.println(jb);  
    jb.setText("New");  
    jb.setCounter(jb.getCounter() + 4);  
    System.out.println(jb);  
}
```

Groovy -> Groovy

```
GrooBean b = [text: 'Text', counter: 1]  
println b  
b.text = 'New'  
b.counter += 4  
println b
```

Groovy -> Java

```
JavaBean jb = new JavaBean('Text', 1)  
println jb  
jb.text = 'New'  
jb.counter += 4  
println jb
```

Groovy syntax:

`b.getProp() => b.prop`

`b.setProp(v) => b.prop = v`

Właściwości związane i ograniczane

Związane -> przy zmianie właściwości są powiadamiani słuchacze,
ograniczone -> słuchacz monitoruje zmiany i może je wetować.

```
import groovy.beans.*  
import java.beans.*
```

Adnotacje!

```
class Bean {  
    @Bindable def text = 'a'  
    @Bindable @Vetoable def count = 1  
}
```

PropertyChangeListener
PropertyChangeEvent

```
Bean b = new Bean()  
b.propertyChange = {  
    println 'Change from ' + it.oldValue + ' to ' + it.newValue  
}  
b.vetoableChange = {  
    if (it.newValue > 5)  
        throw new PropertyVetoException('Change prohibited', it)  
}  
b.text = b.text + 'a'
```

```
try {  
    6.times { b.count++ }  
} catch (PropertyVetoException exc) {  
    println exc.msg + ' for value ' +  
        exc.propertyChangeEvent.newValue  
}
```

```
Change from a to aa  
Change from 1 to 2  
Change from 2 to 3  
Change from 3 to 4  
Change from 4 to 5  
Change prohibited for value 6
```

Definiowanie operatorów

Groovy umożliwia definiowanie (przeciążanie) operatorów w klasach. Operatory są definiowane jako metody o specjalnych nazwach.

Operator	Metoda	Operator	Metoda
+	a.plus(b)	a[b]	a.getAt(b)
-	a.minus(b)	a[b] = c	a.putAt(b, c)
*	a.multiply(b)	a in b	b.isCase(a)
/	a.div(b)	<<	a.leftShift(b)
%	a.mod(b)	>>	a.rightShift(b)
**	a.power(b)	>>>	a.rightShiftUnsigned(b)
	a.or(b)	++	a.next()
&	a.and(b)	--	a.previous()
^	a.xor(b)	+a	a.positive()
as	a.asType(b)	-a	a.negative()
a()	a.call()	~a	a.bitwiseNegate()

Definiowanie operatorów - przykład

```
class Account {  
    def id  
    def balance = 0.0  
  
    Account plus(BigDecimal money) {  
        balance += money  
        return this  
    }  
  
    Account minus(BigDecimal money) {  
        balance -= money  
        return this  
    }  
  
    String toString() { "$id $balance" }  
}  
  
Account a1 = [ id: 'x0001', balance: 1000 ]  
Account a2 = [ id: 'x0002', balance: 2000 ]  
a1 += 100  
a2 = a2 - 1000  
println "$a1, $a2" // x0001 1100, x0002 1000
```

Adnotacje w Groovy = transformacje AST

groovy.lang

Category
Delegate
Grab
GrabConfig
GrabExclude
GrabResolver
Grapes
Immutable
Lazy
Mixin
Newify
PackageScope
Singleton
...

groovy.transform

AutoClone
AutoExternalize
Canonical
EqualsAndHashCode
Field
Immutable
IndexedProperty
InheritConstructors
PackageScope
Synchronized
ToString
TupleConstructor
WithReadLock
WithWriteLock
...

Ćwiczenie: odnaleźć i zapoznać się z dokumentacją wszystkich gotowych transformacji AST

Transformacje AST - przykład 1

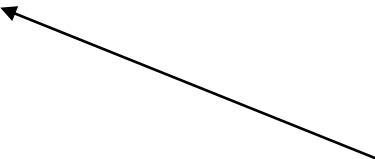
```
import groovy.transform.*           // Na podstawie dokumentacji Groovy API
```

```
@TupleConstructor
@ToString(includeNames=true)
class Conference {
    final title
    @Delegate final Date when
    final where
}
```

Uwaga na final
- wymaga konstruktora
daje go nam adnotacja
@TupleConstructor

```
Conference grConf = ['Groovy', new Date()+1, 'Warsaw' ]
println grConf
Conference scalaConf = [ 'Scala', Date.parse('yyyy-MM-dd', '2018-06-01'),
    'Madrid' ]
println scalaConf
println scalaConf.after(grConf.when)
```

@Delegate generuje
metody delegujące
wywołania do klasy
określonej przez pola



```
Conference(title:Groovy, when:Fri Nov 24 11:14:49 CET 2017, where:Warsaw)
Conference(title:Scala, when:Fri Jun 01 00:00:00 CEST 2018, where:Madrid)
true
```

Transformacje AST – przykład 2

```
/* Przykład na podstawie dokumentacji Groovy API */
/* Uwaga: adnotacja @mixin jest zdezaktualizowana */
```

```
class CollegeStudent {
    def schedule = []
    def addLecture(String lecture) { schedule << lecture }
    def getLectureSchedule() { schedule }
}
```

```
class Worker {
    def schedule = []
    def addMeeting(String meeting) { schedule << meeting }
    def getWorkSchedule() { return schedule }
    public String toString() { schedule }
}
```

```
@Mixin([CollegeStudent, Worker])
```

```
class WorkingStudent {
    public String toString() {
        "working student\nLectures: $lectureSchedule\nJobTasks: $workSchedule"
    }
}
```

```
WorkingStudent ws = new WorkingStudent()
```

```
ws.with {
    addMeeting('Performance review with Boss')
    addLecture('Learn about Groovy Mixins')
    println lectureSchedule
    println workSchedule
    println '-----'
    println schedule // z ostatniej klasy na liście
    println mixedIn[CollegeStudent].schedule
    println mixedIn[Worker].schedule
}
println '-----'
println ws
println ws instanceof Worker
println (ws as Worker)
```

@Mixin – kombinacja metod z różnych klas (prawie wielodziedziczenie)

Zobaczmy też ciekawe słówko 'with', pozwalające unikać powtarzania referencji przy odwołaniach do składowych klasy.

Wynik

```
[Learn about Groovy Mixins]
[Performance review with Boss]
-----
[Performance review with Boss]
[Learn about Groovy Mixins]
[Performance review with Boss]
-----
working student
Lectures: [Learn about Groovy Mixins]
JobTasks: [Performance review with Boss]
false
[Performance review with Boss]
```

Traits

Adnotacja @Mixin została zdeaktualizowana od wersji Groovy 2.3 i generalnie zastąpiona przez *traits* (ale tzw. **runtime mixins** nadal są aktualne – więcej w materiale o metaprogramowaniu).

Trait to kompozycja właściwości i zachowań, ale w odróżnieniu od zwykłych klas *trait* musi być implementowane w innej klasie.

Przykład:

```
trait Person {  
    def name  
    def whoIs() { "$name" }  
}
```

```
trait Learning {  
    def learn() { 'learning Groovy' }  
}
```

```
trait Dancing {  
    def dance() { 'dancing samba' }  
}
```

```
class Student implements Person, Learning, Dancing {}
```

```
Student s = [name: 'Serapion']  
println s.whoIs() + ' is ' + s.learn() + ' and ' + s.dance()  
// Serapion is learning Groovy and dancing samba
```

[Zobacz](#)
[Więcej nt traits](#)

Specjalne nazwy metod

W Groovy metody mogą mieć dowolne nazwy, złożone np ze spacji i znaków specjalnych. Identyfikatory metod i ich wywołania powinny mieć formę literałów napisowych np.:

```
def 'o której będzie wykład?'() {  
    '12:15'  
}  
def godz = 'o której będzie wykład?'()  
println "Wykład będzie o $godz"
```


Dynamiczne wywołania metod

Metody można wywoływać podając ich nazwy (jako napisy) + ew. argumenty. Do takich wywołań stosowana jest składnia GString:

```
def 'która godzina?'() {  
    'jedenasta'  
}
```

```
def ask = { s -> javax.swing.JOptionPane.showInputDialog(s) }  
def mname = ask 'Nazwa metody'  
println 'wołam metodę: ' + mname  
def napis = "$mname"()  
println napis  
mname = ask 'Jakiej metody z klasy String użyć?'  
println 'wołam metodę: ' + mname  
println napis."$mname"()
```

```
// po wprowadzeniu 'która godzina?' a potem toUpperCase dostaniemy:  
wołam metodę: która godzina?  
jedenasta  
wołam metodę: toUpperCase  
JEDENASTA
```

Metody jako domknięcia

Każdą metodę można przekształcić w domknięcie używając `.&` i stosować wszędzie tam, gdzie można zastosować domknięcia (np. przekazując do wykonania innej metodzie).

```
def transform(kod, ...args) {  
    kod(args)  
}
```

```
String s = 'napis'
```

```
def mref = s.&reverse  
println mref.getClass()  
println transform(mref)  
println transform(s.&toUpperCase)  
println transform(s.&getAt, 1)
```

```
// Wynik:
```

```
class org.codehaus.groovy.runtime.MethodClosure  
sipan  
NAPIS  
a
```

Groovy - Java główne różnice

Średnik (J: obligatoryjny, G: opcjonalny)

Nawiasy w wywołaniu metod z argumentami (J: obligatoryjne, G: opcjonalne)

Literały napisowe (J: tylko cudzysłów, G: kilka form)

Porównywanie zawartości obiektów:

- w Javie używaj tylko equals (op. == porównuje referencje)

- w Groovy używaj == (porównanie referencji - metoda **is(..)**)

Dostęp do właściwości

- w Javaie - gettery and settery

- w Groovy - za pomocą kropki lub gettery and settery

Instrukcja return (J: obligatoryjna, G: opcjonalna)

Warunki (J: boolean, G: nie tylko)

Switch (J: ubogie, G: rozbudowane możliwości)

Obsługa wyjątków (J: obligatoryjna dla kontrolowanych, G: nie)

+ Groovy ma wiele rzeczy nieobecnych w Javie:

domknięcia, wieloprzypisania, przeciążanie operatorów, łatwość metaprogramowania.