

Groovy: domknięcia

© Krzysztof Barteczko, PJWSTK 2012 - 2017


Domknięcia

Domknięcie jest fragmentem kodu - **anonimową** funkcją, która może mieć argumenty, zwraca wynik i ma dostęp do wszystkich zmiennych widocznych w otaczającym kontekście (klasie, metodzie, blokach).

{ [closureArguments->] statements }

Tak zapisywane fragmenty kodu można przypisywać zmiennym, przekazywać funkcjom (metodom) i zwracać z funkcji (metod).

```
def add = { x, y -> return x + y }  
def mul = { x, y -> return x*y }  
  
println add(1,5)  
println add.call(1,5)  
  
println op( 7, 8, add)  
println op(7, 8, mul)  
  
def op(x, y, closure) {  
    return closure(x, y)  
}
```



```
6  
6  
15  
56
```

Argumenty domknięć


(1..N). Jednoargumentowe domknięcia nie wymagają deklarowania parametru, a wartość przekazanego argumentu dostępna jest w specjalnej zmiennej **it**. Jeśli wołanie bez arg -> parametr = null.

```
def ok = { println 'Ok' }
def p = { println it }
def pow = { x,y=2 -> println x**y }

def sum = { ... num ->
    sum = 0
    for (n in num) sum += n
    println sum
}

def info = { map ->
    for (en in map) println "${en.key}: ${en.value}"
}

ok()
p 'Groovy'
pow 2
pow 2,4
sum 1,2,3
sum 10,11
info Language: 'Groovy', version: '1.7'
```

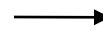


```
Ok
Groovy
4
16
6
21
Language:
Groovy
version: 1.7
```

Wyniki domknięć

Wykonanie domknięcie zawsze zwraca wynik albo na skutek wykonania instrukcji `return` albo jako wynik ostatniej wykonanej w ciele domknięcia instrukcji. Jeśli wynikiem jest *void* zwracana jest wartość *null*.

```
def dbl = { it*2 }  
def max = { x, y -> x > y ? x : y }  
def p = { println it }  
def c = { if (it > 10) return; it }  
  
println dbl(2)  
println max(1,100)  
println p('Some text')  
println c(11)  
println c(7)
```



```
4  
100  
Some text  
null  
null  
7
```

Przekazywanie domknięć funkcjom

Domknięcie można zdefiniować (zadeklarować) **ad hoc** bezpośrednio na liście argumentów wywołania funkcji (metody). Jeśli jest ostatnim argumentem to można zastosować wygodną składnię.

```
func(arg1, arg2, ... argN) {  
    closure_body  
}
```

```
def x = 0  
def incr = { x++ }  
  
op(3, incr)  
println x  
op(3, { x++ } )  
println x  
op(3) { x++ }  
println x  
  
def op(n, closure) {  
    for(i in 1..<n) closure()  
    return closure()  
}
```



3
6
9

Kontekst (zakres) domknięcia

Zakres domknięcia - widoczne w momencie deklaracji zmienne (i inne identyfikatory) z otaczających bloków. W momencie deklaracji następuje związanie tych zmiennych z domknięciem, a przy wykonaniu domknięcie ma żywy dostęp do ich aktualnych w momencie wykonania wartości i może je zmieniać.

```
def x = 0
def incr = { x++ } ← Moment deklaracji. Kod nie jest wykonywany
println x // 0
x = 10
println x
incr() ← Moment wykonania. Domknięcie zna aktualną
println x // 11
```

Zasadnicza różnica!

```
def x = 0
```

```
op(10) { x++ }  
println x    // 10
```

```
def op(n, closure) {  
  // println x  
  for(i in 1..<n) closure()  
  return closure()  
}
```

Funkcja nic nie wie o x (błąd na czerwono)

Ani o tym co robi domknięcie.

Domknięcie ma dostęp do x i jego aktualnych wartości i może je zmieniać

Domknięcia: iteracje liczbowe

```
def txt = ''
7.times {
    txt += "Groovy $it\n"
}
print txt

char c = 'A'
1.upto(10) { print c++ }
println()

def z = '*'
5.downto(1) {
    println z * it
}
```

```
Groovy 0
Groovy 1
Groovy 2
Groovy 3
Groovy 4
Groovy 5
Groovy 6
ABCDEFGHIJ
*****
****
***
**
*
```

Operacja duplikacji (powtórzenia) napisu

Domknięcia: iteracje na obiektach

Dla każdego obiektu, dopuszczającego iteracje są metody:

`each (Closure)` // it == kolejny element Liczba arg. zależy od def.
`eachWithIndex (Closure)` // elt, index -> domknięcia

```
def list = [1,2,3]
list.each { print it }
println()
list.eachWithIndex { e, i -> list[i] *= 2 }
println list
def map = [x: 100, y: 200, w: 300, h: 500 ]
map.each { println it.key + ' ' + it.value }
map.each { k, v -> println "$k = $v" }
map.eachWithIndex { entry, i -> println "[${i}] $entry" }
map.eachWithIndex { k, v, i -> println "(${i}) $k = $v" }
String s = 'Groovy'
def outList = []
s.each { outList << it }
println outList
(1..5).each { print it + ' ' }
def d1 = new Date(), d2 = d1 + 3
print '\n' + (d1.month+1) + ' /'
(d1..d2).each { print ' ' + it.date }
```

```
123
[2, 4, 6]
x 100
y 200
w 300
h 500
x = 100
y = 200
w = 300
h = 500
[0] x=100
[1] y=200
[2] w=300
[3] h=500
(0) x = 100
(1) y = 200
(2) w = 300
(3) h = 500
[G, r, o, o, v, y]
1 2 3 4 5
8 / 12 13 14 15
```

Domknięcia jako klasyfikatory w *switch*

```
def list2 = []  
def list3 = []  
def other = []
```

Domknięcie jako klasyfikator



```
(1..21).each { num ->  
  switch(num) {  
    case { it%2 == 0 } : list2 << num; break  
    case { it%3 == 0 } : list3 << num; break  
    default: other << num  
  }  
}  
println "Multiple of 2: $list2"  
println "Multiple of 3: $list3"  
println "Other: $other"
```

//Wynik:

```
Multiple of 2: [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]  
Multiple of 3: [3, 9, 15, 21]  
Other: [1, 5, 7, 11, 13, 17, 19]
```

Domknięcia: nie ma globalnego return

```
list = 'a b c d '.tokenize()  
func1()  
func2()
```

```
def func1 () {  
    list.each {  
        if (it == 'b') return  
        else println it  
    }  
}
```

// Workaround

```
def func2 () {  
    try {  
        list.each {  
            if (it == 'b') throw new Exception()  
            else println it  
        }  
    } catch (Exception exc) { return }  
}
```

Na konsoli:

```
a  
c  
d  
a
```

Przekształcanie funkcji w domknięcia

```
[def] var = &func_name
```

```
def function(arg) {  
    println arg  
}  
def f = this.&function  
f('A')
```

```
def sysprop = System.&getProperty // getProperty(String) klasy System  
println sysprop('file.separator')
```

Wynik:

```
A  
\
```

Zatem możemy normalne funkcje (metody) przekazywać jako argumenty i zwracać jako wyniki.

Zwijanie domknięć - currying

Przekształcenie funkcji $F: X \times Y \rightarrow R$ w funkcję $G: X \rightarrow (Y \rightarrow R)$

`closure.curry(a1, a2, .. aN)` daje domknięcie z ustalonymi pierwszymi N argumentami.

`closure.rcurry(a1, a2, .. aN)` daje domknięcie z ustalonymi ostatnimi N argumentami.

`closure.ncurry(P, a1, a2, .. aN)` daje domknięcie z ustalonymi N argumentami poczynając od pozycji P na liście argumentów

```
// Groovy release notes
// left currying
def multiply = { a, b -> a * b }
def doubler = multiply.curry(2)
println doubler(10)

// right currying
def divide = { a, b -> a / b }
def halver = divide.rcurry(2)
println halver(20)

// currying n-th parameter
def joinWithSeparator = { one, sep, two ->
    one + sep + two
}
def joinWithComma = joinWithSeparator.ncurry(1, ', ')
println joinWithComma('a', 'b')
```

Wynik:

20

10

a, b

Currying = poproszę funkcję (1)

```
def tagIt = { data, tagKind ->
  "<$tagKind>$data<\\$tagKind>"
}

def bold = tagIt.rcurry('bold')
def italic = tagIt.rcurry('italic')

println bold("Kowalski") + italic("Jan")

//Wynik
<bold>Kowalski<\bold><italic>Jan<\italic>
```

Currying = poproszę funkcję (2)

```
def fmoney = String.&format.curry('%2f') // String.format(...)
def fdate = String.&format.curry('%tF') // zob. opis format w JDK
def dzis = new Date()

println 'Data zakupu ' + fdate(dzis) + ', koszt: ' + fmoney(3.21111)

// wynik
Data zakupu 2017-09-29, koszt: 3,21
```

Kompozycja domknięć

Funkcja złożona: $f(g(h(x)))$

```
// from release notes
def plus2 = { it + 2 }
def times3 = { it * 3}

def times3plus2 = plus2 << times3 // operator << komponuje domknięcia
def plus2times3 = times3 << plus2

println times3plus2(2)
println plus2times3(2)

times3plus2 = times3 >> plus2 // można i w drugą stronę
println times3plus2(2)

// wynik
8
12
8
```


Currying + kompozycja

```
def tagIt = { data, tagKind ->
  "<$tagKind>$data<\\$tagKind>"
}

def bold = tagIt.rcurry('bold')
def italic = tagIt.rcurry('italic')
def bi = italic >> bold

println bi("Kowalski Jan")
```

Wynik:

```
<bold><italic>Kowalski Jan<\italic><\bold>
```

Currying + kompozycja

```
def discount = 0.1
def rate = 0.2

def multiply      = { x, y -> x * y }
def priceWithTax = multiply.curry(1 + rate)
def priceWithDiscount = multiply.curry(1 - discount)

def netPrice = priceWithDiscount << priceWithTax
println netPrice(100)  // wynik: 108.00
```

Trampolina

„When writing recursive algorithms, you may be getting the infamous stack overflow exceptions, as the stack starts to have a too high depth of recursive calls. An approach that helps in those situations is by using Closures and their new trampoline capability.

Closures are wrapped in a TrampolineClosure. Upon calling, a trampolined Closure will call the original Closure waiting for its result. If the outcome of the call is another instance of a TrampolineClosure, created perhaps as a result to a call to the trampoline() method, the Closure will again be invoked. This repetitive invocation of returned trampolined Closures instances will continue until a value other than a trampolined Closure is returned. That value will become the final result of the trampoline. That way, calls are made serially, rather than filling the stack.”

Groovy release notes

To się też nazywa: „Declarative tail-call optimization” (rekurencja ogonowa)

Rekurencja bez trampoliny

```
// Funkcja z Groovy release notes
factorial1 = { int n, BigDecimal accu = 1 ->
    counter++
    if (n < 2) return accu
    factorial1(n - 1, n * accu)
}
```

```
counter = 0
try {
    def f = factorial1(1000)
    println f.toString().size()
    println f
} catch(Throwable exc) {
    println ''+exc
    println counter
}
```

Wynik:

```
java.lang.StackOverflowError // przepełnienie stosu
354                          // tyle było wywołań
```

Trampolina - przykład z rel. notes.

```
counter = 0
factorial2 = { int n, def accu = 1G ->
  counter++
  if (n < 2) return accu
  factorial2.trampoline(n - 1, n * accu)
}
```

Kluczowe: rekurencja
ogonowa, czyli ostatnia
operacja w funkcji
wywołanie samej siebie
lub zwrócenie wyniku

```
factorial2 = factorial2.trampoline()

BigDecimal fac1000 = factorial2(1000)
String s = fac1000.toString()
println 'Liczba wywołań: ' + counter
println 'Liczba cyfr w wyniku: ' + s.size()

// Liczy się i to szybko
// Wynik:
Liczba wywołań: 1000
Liczba cyfr w wyniku: 2568
```

Adnotacja @TailRecursive

Dzięki tej adnotacji, stosowanej wobec metod (funkcji) uzyskujemy automatycznie odpowiednią sekwencję trampolinowania domknięcia, uzyskanego z funkcji (łatwiejszy kod)

// przykład z <http://mrhaki.blogspot.com>

// rekurencyjne liczenie rozmiaru listy

```
def sizeOfList(list, counter = 0) {  
    if (list.size() == 0) {  
        counter  
    } else {  
        sizeOfList(list.tail(), counter + 1)  
    }  
}  
println sizeOfList(1..10000) // 10000
```

zob. co robi tail() w Groovy JDK

[Zob. dokumentację](#)

Memoizacja (memoization)

The return values for a given set of Closure parameter values are kept in a cache, for those memoized Closures. That way, if you have an expensive computation to make that takes seconds, you can put the return value in cache, so that the next execution with the same parameter will return the same result – again, we assume results of an invocation are the same given the same set of parameter values.

There are three forms of memoize functions:

- the standard `memoize()` which caches all the invocations
- `memoizeAtMost(max)` call which caches a maximum number of invocations
- `memoizeAtLeast(min)` call which keeps at least a certain number of invocation results
- and `memoizeBetween(min, max)` which keeps a range results (between a minimum and a maximum)

Memoizacja - przykład

```
long start
long count = 0
def startTimer = { start = System.currentTimeMillis() }
def elapsed = { System.currentTimeMillis() - start }

fib = { n ->
    count++
    if (n < 2) n
    else fib(n - 1) + fib(n - 2)
}

startTimer()
println fib(34)
println 'Czas = ' + elapsed()/1000 + ' sek. Wolań: ' + count

fib = fib.memoize()
count = 0
startTimer()
println fib(34)
println 'Czas = ' + elapsed()/1000 + ' sek. Wolań: ' + count
```

```
Wynik na konsoli:
5702887
Czas = 15.125 sek. Wolań: 18454929
5702887
Czas = 0 sek. Wolań: 35
```


Adnotacja @Memoized

Przekształca funkcję w domknięcie i stosuje metodę memoize().
Łatwiejszy, bardziej naturalny kod.

```
import groovy.transform.*;
```

```
@Memoized
```

```
def fib(n) {  
    if (n < 2) n  
    else fib(n - 1) + fib(n - 2)  
}
```

```
println fib(34) // wynik jak poprzednio w milisekundy gotowy
```



[Zob. dokumentację](#)