

# **Podstawy Groovy: napisy, listy, mapy, sterowanie**

© Krzysztof Barteczko, PJWSTK 2012-2017

# Rodzaje napisów w Groovy

'aa'

"bb"

'aaaaa "oo" bb'

"bbbbbb 'cd' ef"

"v = \$v"

"Today = \${new Date()}"

"x[1] = \${x[1]}"



Gdy "..." i \$ - GString z  
substytucją zmiennych i  
wyrażeń

'''first line

second'''

""""multi

line ev. Gstring""""



Napisy wielowierszowe

'Slashy' Strings



/In slashy Strings you can use \ without escaping/

# Rodzaje napisów w Groovy (2)

## Multiline slashy strings:

```
def poem = /  
to be  
or  
not to be  
/
```

## Dollar slashy strings = slashy + GString:

„multi-line GString similar to the slashy string, but with slightly different escaping rules. You are no longer required to escape slash (with a preceding backslash) but you can use '\$\$' to escape a '\$' or '\$/' to escape a slash if needed”.

```
def town = "Warsaw"  
  
def dollarSlashy = $/  
    In $town,  
    today is ${new Date() }  
    normal $ dollar-sign  
    $$town dollar-sign escaped  
    \ backslash  
    / slash  
    $/$$ slash and dollar  
/$  
  
println dollarSlashy
```

```
In Warsaw,  
today is Tue Mar 13 05:05:05 CET 2012  
normal $ dollar-sign  
$town dollar-sign escaped  
\ backslash  
/ slash  
/$ slash and dollar
```

[zobacz w dokumentacji](#)

# Łączenie napisów

Do łączenia napisów służy operator +

```
txt1 = 'Groovy'  
txt2 = 'an agile language'  
res = txt1 + ' is ' + txt2 // Groovy is an agile language
```

Można dołączać inne dane (będą przekształcone na napisy):

```
res = txt1 + ' number ' + 1 // Groovy number 1  
  
now = new Date()  
res = 'Now is: ' + now // Now is: Sun Aug 09 05:24:07 CEST 2009
```

Uwaga:  
String na  
początku!

Użycie cudzysłowu daje GString (substytucja wyrażeń poprzedzonych \$):

```
res = "$txt1 is $txt2" // Groovy is an agile language  
  
res = "Now is: ${new Date()}" // Now is: Sun Aug 09 05:24:07 CEST 2009
```

Jeśli wyrażenie nie jest zmienną  
ujmujemy je w nawiasy klamrowe

# Proste działania na napisach

## A. Metody klasy String z JDK

## B. Dodatkowo Groovy:

```
def s = 'abcdefegh'
println s.size()
println s[0] // indeksowanie
println s[1] // daje znak
println s[-1] // ostatni znak
println s[0..2] // podciąg 0-2
println s[0..<2] // od 0 do 1

println s.reverse() //odwrócenie
println s[-1..0] // można tak
// wyrównanie napisów w polach
println s.padLeft(15, '+')
println s.padRight(15, '+')
println s.center(15, '*')

println s*2 // duplikacja
println s - 'e' // usunięcie
println s - 'abc'
```

**Wyniki** działań z lewej strony:

```
9
a
b
h
abc
ab

hgefedcba
hgefedcba

+++++abcdefegh
abcdefegh+++++
***abcdefegh***

abcdefeghabcdefegh
abcdfeqh
defegh
```


Groovy JDK = rozszerzenie JDK

Zob. [dokumentacja klasy String](#) (głównie metody dziedziczone z CharSequence)

# Konwersje napisy - liczby

Pojedyncze znaki mają swoje liczbowe kody i mogą być traktowane jak liczby:

```
char a = 'A'
int kod = a
println "$a - code: $kod"
char c = '1'
println "$c - code: " + (int) c
kod++
c = (char) kod
println "$c - code: $kod"
a += 1
println a
```



A	- code: 65
1	- code: 49
B	- code: 66
B	

Operator rzutowania

Po to by z napisu (a nie typu char) uzyskać liczbę, potrzebne są dodatkowe środki:

```
s = '65'
```

```
int v
```

```
v = s
```

Błąd wykonania

```
v = s.toInteger() + 1 // Ok, 66
```

**albo:**

```
v = s as Integer
```

**Dostępne są też:**

**s.toDouble()**

**s.toBigDecimal() .... inne**

# Błędy konwersji

Gdy napis nie daje się potraktować jako liczba odpowiedniego typu występuje wyjątek `NumberFormatException`. Można go oczywiście obsłużyć (jak w Javie), ale w Groovy istnieje możliwość wcześniejszego sprawdzenia czy napis może być traktowany jako liczba odpowiedniego typu.

```
def s1 = '10'  
def s2 = '1.1'
```

```
println s1.isInteger()  
println s2.isDouble()  
println s2.isBigDecimal()  
println s1.isDouble()  
println s1.isBigDecimal()  
println s2.isInteger()  
println s1.isNumber()  
println s2.isNumber()
```

Result:

```
true  
true  
true  
true  
true  
false  
true  
true
```



Metody sprawdzania typu

# Listy - wprowadzenie

Lista - zestaw elementów, które mają określone pozycje w zestawie i mogą się powtarzać.

Utworzenie listy w programie:

```
list1 = ['A', 'B', 'C']
```

```
list2 = [1, 2, 3, 4, 5]
```

```
list3 = [1, 'A', 2, 'B', new Date()]
```

```
elist = [] // pusta lista
```

Takie listy są typu:

Collection, List i ArrayList.

Dostęp do elementów (indeksowanie od 0):

```
println list1[0] // A
```

```
list3[1] = 2 // teraz [1,2,2,B ...]
```

Liczba elementów na liście: `list1.size()` // == 3

Ost. element: `list1[-1]` // C

Odwracanie: `list1.reverse()` lub `list1[-1..0]`

Sublisty: `list[1..2]` // [ B, C ]

`list[-2..-1]` // dwa ostatnie znaki

Dodawanie elementów do listy:

```
elist << 77 << 81 // teraz [77, 81]
```

```
list1 += 'A' // teraz [A, B, C, A]
```

Usuwanie elementów z listy:

```
list1 -= 'A' // teraz [B, C]
```

```
list2.remove(1) // teraz [1, 3, 4, 5]
```

[Dokumentacja składni](#)

Usuwa dany element

Usuwa element  
o podanym indeksie



# Wieloprzypisania

W jednej instrukcji przypisanie na wiele zmiennych  
(lista zmiennych) = dowolny obiekt z operatorem indeksowania

(a, b) = [1, 2]

(a, b) = [b,a] // swap

(a, b) = 'ab'



[Szczegółowa dokumentacja](#)

# Rozbiór tekstów

```
def s1 = 'Groovy, Java - ok'
```

```
def list = s1.tokenize()
```

Separator = białe znaki  
' \t\n\r\f'

```
println list
```

```
list = s1.tokenize(' , -')
```

Separator = podane

```
println list
```

Wynik:

```
[Groovy,, Java, -, ok]
```

```
[Groovy, Java, ok]
```

[Szczegółowa dokumentacja](#)

# Łączenie elementów kolekcji w napisy

## **String join(String separator)**

Concatenates the toString() representation of each item in this collection, with the given String as a separator between each item.

```
String txt
def list = [1,2,3,4]
txt = list.join(' - ')
println txt
```

```
txt = ' a b      c d'
list = txt.tokenize()
txt = list.join(' ')
println txt
```

Ten sam efekt:

```
txt = txt.tokenize().join(' ')
```

### **Result:**

```
1 - 2 - 3 - 4
a b c d
```

# Interakcja z programem

## **Wprowadzania danych:**

### **z konsoli za pomocą skanera (klasa Scanner)**

```
input = new Scanner(System.in).next...()
```

### **z użyciem dialogu wejściowego**

```
input = JOptionPane.showInputDialog('Message')  
(wynikiem jest wprowadzony napis lub null, jeśli anulowano  
dialog)
```

## **Pokazywanie wyników w okienku komunikatów**

```
JOptionPane.showMessageDialog(null, string)
```

# Interakcja – przykład 1

```
import static javax.swing.JOptionPane.*

(kwota, ods) = showInputDialog('Podaj kwote i oprocentowanie').tokenize()
println "Kwota: $kwota"
println "Oprocentowanie: $ods"

kwota = kwota.toBigDecimal()
ods = ods.toBigDecimal()

println 'Po dodaniu odsetek kwota wynosi ' + (1+ods)*kwota
```

# Interakcja - przykład 2

```
import static javax.swing.JOptionPane.*
```

```
input = showInputDialog("Enter phrase")
```

```
data = input.tokenize() ← Rozbicie tekstu na słowa  
println data  
println "Phrase contains ${data.size()} word(s)"  
println 'Enter line in console'  
input = new Scanner(System.in).nextLine()  
println input
```

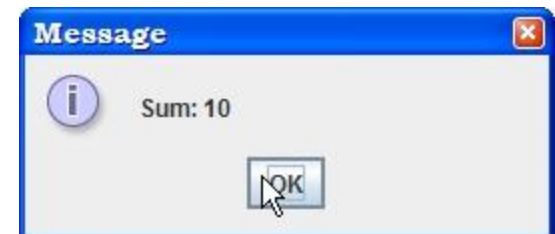
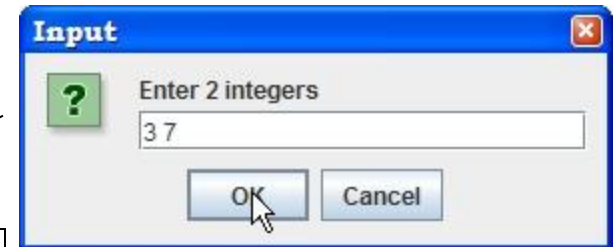
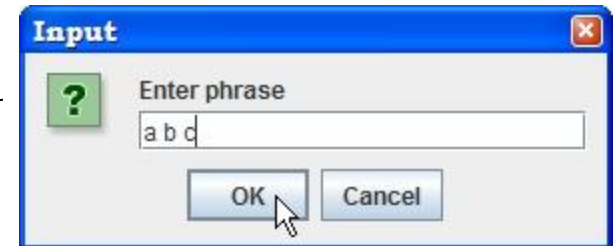
```
input = showInputDialog("Enter 2 integers")
```

```
scan = new Scanner(input) ← Skaner nałożony na String
```

```
sum = scan.nextInt() + scan.nextInt() ← nextInt pobiera kolejny symbol i  
showMessageDialog(null, "Sum: $sum")      przekształca go w liczbę całkowitą
```



	Console
[a, b, c]	
Phrase contains 3 word(s)	
Enter line in console	
Groovy lang	
Groovy lang	



# Mapy - wprowadzenie

Mapa: zestaw par klucz-wartość

Tworzenie w programie:

```
map1 = [:] // pusta mapa
```

```
map2 = [ Joan: '622-130-170', Steve: '677-190-278']
```

```
map3 = [:].withDefault { 10 } // z domyślnymi wartościami
```

Dostęp:

```
map2.Joan // 622-130-170
```

```
map2['Joan'] // 622-130-170
```

```
map2.Dave // null
```

```
name = 'Steve'
```

```
map2[name] // 677-190-278
```

```
map1['x'] = 7
```

```
map1.y = 23
```

```
map1.x + map1.y // 30
```

```
map3.x = 10
```

```
map3.x + map3.y // 20
```

Takie mapy są typu:

Map i LinkedHashMap

Zbiór kluczy:

map.keySet()

Zbiór wartości:

map.values()

Kolekcja wejść:

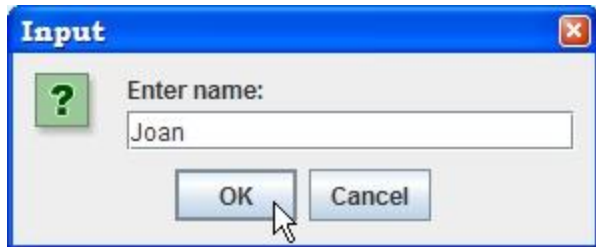
map.entries()



[Zob. w dokumentacji](#)

# Interakcja - przykład 3

```
import static javax.swing.JOptionPane.*  
map2 = [ Joan: '622-130-170', Steve: '677-190-278']  
name = showInputDialog("Enter name:")  
tel = map2[name]  
showMessageDialog(null, "$name - tel. $tel")
```





# Sterowanie

if (..) { }

if (...) { }  
else { }

Możliwe własne konstrukcje!

while(...) { }  
for (init; cond; upd) { }

for (typ x : obj) { }  
for (x in obj)



[Zob. w dokumentacji](#)

operator in  
switch(...) ...

Operator warunkowy  
Elvis-operator

try { } catch { } finally { }

throw exc

# Warunki

W Groovy warunkiem może być wyrażenie dowolnego typu (nie void).

W momencie testowania warunku w takich instrukcjach jak *if* i *while* wyrażenie jest przekształcane do wartości boolowskiej (*true* lub *false*) wg. następującego schematu:

Typ wyrażenia	Wynikiem jest true gdy:
Boolean	wyrażenie ma wartość true
Matcher	znaleziono dopasowanie
Collection	kolekcja (np. lista) nie jest pusta
Map	mapa nie jest pusta
String, GString	napis nie jest pusty
Number, Character	wartość nie jest zero
dowolny inny	referencja do obiektu nie jest null

Uwaga: w Javie warunkiem może być tylko wyrażenie typu boolean.

# Porównania

Operatory służące do porównań:

>, <, >=, <=      wołana jest metoda a.compareTo(b)

==, !=              wołana jest metoda a.equals(b)

```
def txt1 = 'Groovy', txt2 = 'Java'
def x = 1, y = 10
def list = ['a', 'b', 'c']
def elist = ['a', 'b', 'c']
max = y
if (x > y) max = x
println max
txt3 = 'G'
txt3 += 'roovy'
println txt2 > txt1
println txt1 == txt3
println list == elist
```

Wyniki porównań zależą od definicji metod equals i compareTo w klasach obiektów.

```
10      // bo x.compareTo(y) zwraca false
true    // bo 'Java'.compareTo('Groovy') zwraca wartość > 0
true    // bo zawartość obiektów wskazywanych przez txt1 i txt2 jest taka sama
true    // bo list i elist ma tę samą zawartość
```

# Wyrażenia warunkowe

**Operator warunkowy ?:** ma trzy argumenty - wyrażenia i stosowany jest do konstrukcji wyrażenia warunkowego w następujący sposób:

`e1 ? e2 : e3` // jeśli e1 daje true wynikiem jest e2, inaczej e3

**Elvis-operator** (skrót dla niektórych sytuacji)

`e1 ?: e2` // wynik: jeśli e1 daje true (np. !null) e1, inaczej e2

```
import static javax.swing.JOptionPane.*;

name = showInputDialog('Enter name')
userName = name ? name : 'Anonymous' // ternary
println "User: $userName"

name = showInputDialog('Enter name') ?: 'Anonymous'
println "User: $name"
```

# Zakresy

Obiekty typu Range określają zakresy. Wprowadzamy je za pomocą operatorów `..` lub `..`

Przykłady:

```
1..10 // zestaw liczb od 1 do 10
```

```
1..<10 // zestaw liczb od 1 do 9
```

```
10..1 // zakres odwrócony (10, 9, 8, ... 1)
```

```
'a'..'c' // litery a, b, c
```

```
d1 = new Date()
```

```
d2 = d1 + 7
```

```
d1..d2 // daty od dziś do za tydzień
```

# Operator in

Wyrażenie:

`e in ob`

użyte jako warunek (nie w iteracyjnym for) ma wartość true jeśli w klasie obiektu ob zdefiniowana metoda `isCase()` zwraca true.

```
def r1 = 1..10
def r2 = 'a'..'c'
def d = new Date()
def r3 = d..d + 7
def l = [1, 3, 7]
def map = [ a: 1, b: 2]
def s = 'xaz'
println 3 in r1
println 11 in r1
println 'b' in r2
println 'z' in r2
println d + 3 in r3
println d + 10 in r3
println 3 in l
println 'b' in map // isCase sprawdza zawartość keySet
println 'xaz' in s // isCase == equals
// ale:
println 'x' in s
```

true
false
true
false
true
false
true
true
true
false

# Instrukcja switch

```
switch(candidate) {  
    case classifier1 :  
        code  
        [ break ]
```

...

```
case classifierN : code; [ break]  
default : code
```

```
}
```

Do klasyfikatora której etykiety *case* pasuje kandydat? Tam przejdzie sterowanie. Jeśli nie ma *break* wykonywany jest kod następnej etykiety. Jeśli kandydat nigdzie nie pasuje - wykonywany jest kod etykiety *default*.

Decyduje *isCase(candidate)* z klasy obiektu *case*.

## Przykład:

```
switch(e) {  
    case 1 : println "$e - one"; break  
    case 'Groovy' : println "$e - lang"; break  
    case 'a'..'z' : println "$e - in range"; break  
    case [ 'ala', 'kot' ] : println "$e - in list"; break  
    case [ klucz1: 1, klucz2: 2 ] : println "$e - key in map"; break  
    case BigDecimal : println "$e - BigDecimal"; break  
    default: println "$e - Don't know it"  
}
```

## Wynik:

```
if (e== 1) 1 - one  
if (e == 'Groovy') Groovy - lang  
if (e == 'x') x - in range  
if ( e == 2.3) 2.3 - BigDecimal  
ala - in list  
klucz1 - key in map
```

```
switch (num) {  
    case 1..3 :  
        println 'Small'  
        break  
    case 4..7 :  
        println 'Medium'  
        break  
    case 8..11 :  
        println 'Large'  
        break  
    default:  
        println 'Undefined'  
}
```

# Instrukcja for-each

Ma postać:

**for (*var in iterable*) *ins***

gdzie:

var - nazwa zmiennej

iterable - obiekt iterowalny (np. zakres, lista, mapa, napis)

ins - instrukcja

Działanie:

w każdym kroku pętli kolejny element obiektu iterowalnego (zestawu) jest podstawiany na zmienną var i wykonywana jest instrukcja ins (w której zazwyczaj sięgamy do zmiennej var)

Zgodna z Javą postać tej instrukcji:

**for (*Typ*|def *var* : *iterable*) *ins***

ma takie same działanie, ale wymaga deklaracji zmiennej var

Gotowe iterable:

Kolekcje

Mapy

Zakresy

Napisy



# For-each przykłady

```
rng = 1..10
sum = 0
for (x in rng) sum += x
println sum

list = [ 'a', 'b', 'c' ]
list1 = []
for (elt in list) list1 << elt + 'X'
println list1

for (x in [a:1, b:2, c:3]) {
    println x
    println "$x.key $x.value"
}

low = 'a'
up = 'd'
for (c in up..<low) print c

napis = 'Warszawa'
println()
for (c in napis) print "-$c-"
```

entry w mapie

Z entry można pobrać  
klucz (.key) i wartość  
(.value)



```
55
[aX, bX, cX]
a=1
a 1
b=2
b 2
c=3
c 3
dcb
-W--a--r--s--z--a--w--a--
```