

Wyrażenia regularne

© Krzysztof Barteczko, PJWSTK 2012 - 2017

Pojęcie wyrażeń regularnych

Regularne wyrażenie stanowi opis wspólnych cech (składni) zbioru łańcuchów znakowych
== wzorzec, który opisuje jeden lub wiele napisów, pasujących do tego wzorca.

Wzorzec zapisujemy za pomocą specjalnej składni wyrażeń regularnych.

Najprostszym wzorcem jest po prostu sekwencja znaków, które nie mają specjalnego znaczenia: napis 'a' pasuje do wzorca 'a'

We wzorcach możemy stosować znaki specjalne (tzw. metaznaki) oraz tworzone za ich pomocą konstrukcje składniowe.

Znaki specjalne (metaznaki)

\$	^	.	*	
+	?	[]	
()	{	}	\

Za pomocą znaków specjalnych i tworzonych za ich pomocą bardziej rozbudowanych konstrukcji składniowych opisujemy m.in.

- * wystąpienie jednego z wielu znaków - **klasy znaków**,
- * początek lub koniec ograniczonego ciągu znaków - **granice**,
- * powtórzenia - **kwantyfikatory**,
- * logiczne kombinacje wyrażeń regularnych.

Np.. wyrażenie regularne $[0-9]$ stanowi wzorzec opisujący jeden znak, który może być dowolną cyfrą 0,1,2,...,9. Wzorzec ten opisuje wszystkie napisy składające się z jednej cyfry.

A wyrażenie regularne $a.*z$ (a, kropka, gwiazdka, z) opisuje dowolną sekwencję znaków, zaczynających się od litery a i kończących się literą z. Do wzorca tego pasują np. następujące napisy: "az", "abz", "a x y z".

Do czego używa się wyr. regularnych?

- * stwierdzenia czy dany napis pasuje do podanego przez wyrażenie wzorca,
- * stwierdzenia czy dany napis zawiera podłańcuch znakowy pasujący do podanego wzorca i ew. uzyskania tego podnapisu i/lub jego pozycji w napisie,
- * zamiany części napisu, pasujących do wzorca na inne napisy,
- * wyróżniania części napisu, które są rozdzielane ciągami znaków posuwającymi do podanego wzorca.

Groovy: operator =~

Wyrażenie:

`text =~ regex`

użyte w miejscu warunku zwraca true, jeśli w tekście text znajduje się ciąg znaków pasujący do wzorca regex (podanego jako napis).

```
def text = 'Groovy is cool with regex'
```

```
if (text =~ 'x') println 'x found'
```

```
else println 'x not found'
```

```
if (text =~ 'z') println 'z found'
```

```
else println 'z not found'
```

```
x found
```

```
z not found
```

Groovy: operator ==~

Wyrażenie:

`text ==~ regex`

ma wartość true, jeśli cały tekst text pasuje do wzorca regex.

```
def text = 'Groovy'

if (text ==~ 'Groovy') println 'Groovy'
else println 'No match'
if (text ==~ 'Java') println 'Java'
else println 'No match'
```

```
Groovy
No match
```

Pattern i Matcher

Przed zastosowaniem wzorzec jest kompilowany.

Obiekty klasy **Pattern** reprezentują skompilowane wyrażenia regularne, a obiekty klasy **Matcher** - tzw. silnik wyrażen regularnych, wykonujący operacje dopasowania tekstu lub jego części do skompilowanego wzorca.

Wynikiem wyrażenia:

```
text =~ regex
```

Wyrażenie <code>~regex</code> jest typu <code>Pattern</code>

(jeśli nie występuje ono jako warunek) jest obiekt klasy `Matcher` związany ze skompilowanym wzorcem `regex` i gotowy do wyszukiwania.

```
def matcher = 'ala' =~ 'a'  
println matcher
```

Result:

```
java.util.regex.Matcher[pattern=a region=0,3 lastmatch=]
```

Metody i stany matchera

boolean	<code>find()</code> Przeszukuje łańcuch wejściowy poczynając od początku (lub poprzedniego dopasowania) w poszukiwaniu podłańcucha pasującego do wzorca. Jeśli wzorzec został dopasowany (wynik true), kolejne wywołanie tej metody przeszukuje łańcuch wejściowy poczynając od znaku po ostatnim dopasowanym znaku. Jeśli wzorzec nie został znaleziony metoda zwraca wartość false.
boolean	<code>matches()</code> Sprawdza dopasowanie całego łańcucha wejściowego do wzorca. Zwraca true, jeśli takie dopasowanie występuje i false w przeciwnym razie.

Stany (m.in.): pozycje ostatniego dopasowania (metody `start()` i `end()`)

Metoda `reset()` resetuje stany, a z podanym tekstem ustale dodatkowo nowy tekst do przeszukiwania.

Przykład find()

```
def text = 'Groovy is cool with regex'
def regex = 'o'
def matcher = text =~ regex
show(text, matcher)
text = 'xxxo'
matcher.reset(text)
show(text, matcher)

def show(txt, m) {
    def regex = m.pattern()
    while (m.find()) {
        println "In '$txt' found '$regex' start: ${m.start()} end: ${m.end()}"
    }
}
```

Result:

```
In 'Groovy is cool with regex' found 'o' start: 2 end: 3
In 'Groovy is cool with regex' found 'o' start: 3 end: 4
In 'Groovy is cool with regex' found 'o' start: 11 end: 12
In 'Groovy is cool with regex' found 'o' start: 12 end: 13
In 'xxxo' found 'o' start: 3 end: 4
```

Klasy znaków

Prosta klasa znaków stanowi ciąg znaków ujętych w nawiasy kwadratowe np.

[123abc]

Do takiego wzorca pasuje dowolny z wymienionych znaków.

Jeśli pierwszym znakiem w nawiasach kwadratowych jest ^, to dopasowanie nastąpi dla każdego znaku oprócz wymienionych na liście. Jest to swoista negacja klasy znaków.

Np. do wzorca [^abc] będzie pasował każdy znak oprócz a, b i c.

Możliwe jest także formułowanie zakresów znaków Przykładowe wzorce:

[0-9] - dowolna cyfra,

[a-zA-Z] - dowolna mała i duża litera alfabetu angielskiego.

[a-zA-Z0-9] = dowolna cyfra lub litera

Klasy predefiniowane

.	Dowolny znak
\d	Cyfra: [0-9]
\D	Nie-cyfra: [^0-9]
\s	"Biały" znak: [\t\n\x0B\f\r]
\S	Każdy znak, oprócz "białego": [^\s]
\w	Jeden ze znaków: [a-zA-Z0-9_]
\W	Znak nie będący literą lub cyfrą [^\w]

Slashy strings!

```
def text = ['A 100', 'A B']
def rlist = [ /[A-Z] \d\d\d/, ' /.../' ]

text.each { txt ->
  rlist.each { regex ->
    def msg = 'matches'
    if (!(txt =~ regex)) msg = 'no ' + msg
    println "'$txt' $msg '$regex'"
  }
}
```

Result:

```
'A 100' matches '[A-Z] \d\d\d'
'A 100' no matches '...'
'A B' no matches '[A-Z] \d\d\d'
'A B' matches '...'
```

Klasy Posixowe

<code>\p{Lower}</code>	Mała litera: [a-z]
<code>\p{Upper}</code>	Duża litera: [A-Z]
<code>\p{ASCII}</code>	Dowolny znak ASCII :[\x00-\x7F]
<code>\p{Alpha}</code>	Dowolna litera: [\p{Lower}\p{Upper}]
<code>\p{Digit}</code>	Cyfra: [0-9]
<code>\p{Alnum}</code>	Cyfra bądź litera: [\p{Alpha}\p{Digit}]
<code>\p{Punct}</code>	! " # \$ % & ' () * + , - . / : ; < = > ? @ [\] ^ _ ` { } ~
<code>\p{Graph}</code>	Widzialny znak: [\p{Alnum}\p{Punct}]
<code>\p{Print}</code>	Drukowalny znak: [\p{Graph}]
<code>\p{Blank}</code>	Spacja lub tabulacja: [\t]
<code>\p{Cntrl}</code>	Znak sterujący: [\x00-\x1F\x7F]
<code>\p{XDigit}</code>	Cyfra szesnastkowa: [0-9a-fA-F]
<code>\p{Space}</code>	Biały znak: [\t\n\x0B\f\r]
<code>\p{L}</code>	Dowolna litera (Unicode)
<code>\p{Lu}</code>	Dowolna duża litera (Unicode)
<code>\p{Ll}</code>	Dowolna mała litera
<code>\p{InNazwaBlokuUnicode}</code>	Znak z podanego bloku Unicode

Matcher find() w Groovy

```
def text = 'alabama'  
def matcher = text =~ 'a.a'
```

```
println matcher.size()  
println matcher[0]  
println matcher[1]
```

Matcher jak lista



```
for (match in matcher) println match
```

Domknięcie dostaje
kolejne dopasowania

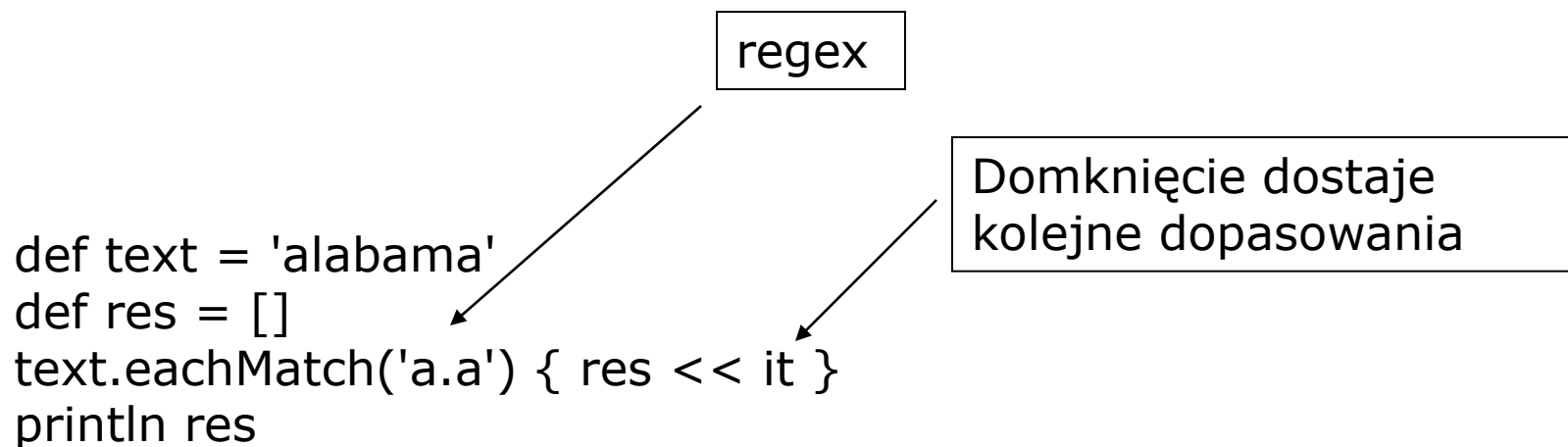


```
def res = []  
matcher.each { res << it }  
println res
```

Result:

```
2  
ala  
ama  
ala  
ama  
[ala, ama]
```

Jeszcze prościej: eachMatch ze String



Result:

[ala, ama]

Kwantyfikatory

Wyrażenia regularne byłyby całkiem nieprzydatne, gdyby nie można było za ich pomocą dopasowywać powtarzających się sekwencji znaków. Do specyfikacji powtórzeń służą kwantyfikatory.

Symbole kwantyfikatorów są następujące i oznaczają:

?	wystąpienie jeden raz lub wcale
*	wystąpienie zero lub więcej razy
+	wystąpienie raz lub więcej razy
{n}	wystąpienie dokładnie n razy
{n,}	wystąpienie co najmniej n razy
{n,m}	wystąpienie co najmniej n ale nie więcej niż m razy

Czego dotyczy wystąpienie?

Kwantyfikator po literale - wymagane jest wystąpienie (liczba wystąpień zależy od kwantyfikatora, w szczególności może być 0) tego literału np. "12a+" oznacza 1, potem 2, następnie wystąpienie znaku 'a' jeden lub więcej razy.

Kwantyfikator po klasie znaków - dotyczy dowolnego znaku z tej klasy. Np. [abc]+ oznacza wystąpienie jeden lub więcej razy znaku a, lub znaku b, lub znaku c.

Kwantyfikator dotyczący dowolnego wyrażenia regularnego X:

(X)symbol_kwantyfikatora

(?:X)symbol_kwantyfikatora

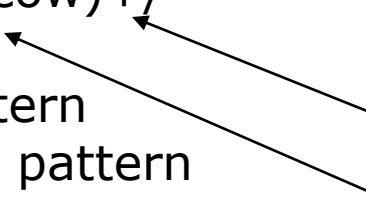
Pierwsza z w/w form składniowych służy też do zapamiętywania tekstu pasującego do wzorca podanego w nawiasach. Druga forma służy wyłącznie grupowaniu, bez zapamiętywania.

Przykład

Wzorzec opisuje jedno lub wiele wystąpień dowolnych ze słów dog, cat, cow (nie rozdzielonych spacjami).

```
def pattern = /(dog|cat|cow)+/
```

```
println 'dogcat' ==~ pattern  
println 'dogdogdog' ==~ pattern  
println 'catcatcow' ==~ pattern
```



Kwantyfikator +
Symbol alternatywy - albo

Result:

```
true  
true  
true
```

Kwantyfikatory zachłanne i wstrzemięźliwe

Zachłanne - konsumują cały tekst wejściowy (i starają się go dopasować). Jeśli to się nie uda, następuje cofanie znak po znaku, aż do uzyskania dopasowania lub jego braku.

Wstrzemięźliwe - rozpoczynają od początku tekstu wejściowego i pobierają znak po znaku szukając dopasowania.

Aby uczynić zachłanny (**greedy**) kwantyfikator:

$?, *, +, (n), (n,), (n,m)$

wstrzemięźliwym (**reluctant**) dodajemy po nim znak ?

Czyli kwantyfikatory wyglądają tak:

$??, *?, +?, (n)?, (n,)?, (n,m)?$

Przykład

```
def text = 'oops foo bar foo'
def pattern = ""

println 'Greedy:'
text.eachMatch ('.*foo') { println it }

println 'Reluctant:'
text.eachMatch ('.*?foo') { println it }
```

Result

```
Greedy:
oops foo bar foo
Reluctant:
oops foo
bar foo
```

Grupy

Użycie nawiasów okrągłych '(' i ')' pozwala zapamiętywać części tekstu pasujące do wzorca podanego w nawiasach. Grupy są kolejno numerowane poczynając od 1. Mówiąc ściślej są numerowane poprzez zliczanie otwartych i zamkniętych nawiasów.

Np. w wyrażeniu (A) (B) (C) mamy trzy grupy o numerach:

- 1 - grupa odpowiadająca wyrażeniu A
- 2 - grupa odpowiadająca wyrażeniu B
- 3 - grupa odpowiadająca wyrażeniu C

a w wyrażeniu ((A)((B)(C)))(D) mamy 6 grup o numerach

- 1 - grupa odpowiadająca wyrażeniu (A)((B)(C))
- 2 - grupa odpowiadająca wyrażeniu A
- 3 - grupa odpowiadająca wyrażeniu (B)(C)
- 4 - grupa odpowiadająca wyrażeniu B
- 5 - grupa odpowiadająca wyrażeniu C
- 6 - grupa odpowiadająca wyrażeniu D

Dostęp do zawartości grup

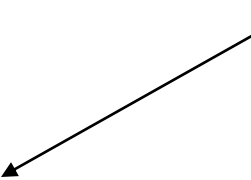
Klasyczny:

```
def txt = 'London    1957  Warsaw 2009'  
def regex = /(\w+)\s+(\d+)/
```

```
def matcher = txt =~ regex
```

```
while (matcher.find()) {  
  for (int i = 0; i <= matcher.groupCount(); i++)  
    println i + ' ' + matcher.group(i)  
}
```

Liczba grup



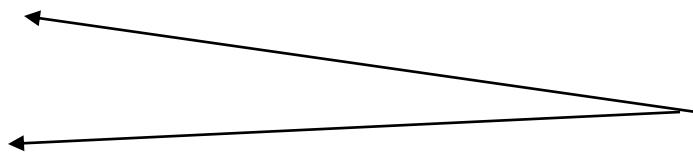
Grupa nr i



Result:

```
0 London    1957  
1 London  
2 1957  
0 Warsaw 2009  
1 Warsaw  
2 2009
```

Grupa 0 - cały
odnaleziony tekst



Dostęp do grup - Groovy

```
def txt = 'London 1957 Warsaw 2009'
def regex = /(\w+)\s+(\d+)/
def matcher = txt =~ regex
```

```
println matcher[0]
println matcher[0][1]
println matcher[0][2]
println matcher[1]
println matcher[1][1]
println matcher[1][2]
```

Jak są grupy - lista grup

Pierwszy indeks - wystąpienie
Drugi indeks - numer grupy

```
def map = [:]
txt.eachMatch(regex) { all, town, date ->
    map[town] = date
}
```

Kolejne wystąpienie (całe)

Pierwsza grupa w nim

Druga grupa w nim

```
map.each { println it.key + ' - ' + it.value }
```

Result:

```
[London 1957, London, 1957]
```

```
London
```

```
1957
```

```
[Warsaw 2009, Warsaw, 2009]
```

```
Warsaw
```

```
2009
```

```
London - 1957
```

```
Warsaw - 2009
```

Granice i flagi

Granice pozwalają na dopasowaniem wzorca w pewnym konkretnym miejscu tekstu

Np.

^ Początek linii

\$ Koniec linii

Sposób interpretacji wyrażenia regularnego można modyfikować za pomocą **flag**.

Np.

(?s) pozwala na dopasowanie metaznaku . (kropka) również do znaku końca wiersza,

(?i) na porównania liter bez uwzględnienia ich wielkości

Przykład - nagłówki <h2> z HTML

Jak krótko:

```
def headers = []  
doc.eachMatch('( ?s)<h2>(.*?)</h2>') { all, txt -> headers << txt }  
headers.each { println it }
```

Nie greedy!!!

doc:

```
<h2>Punkt1</h2>bl  
ablabla  
dlablabla<h2>Punkt  
2</h2>  
<h2>Punkt3</h2>bl  
a
```

Punkt1
Punkt2
Punkt3

Zastępowanie

Matcher umożliwia zastępowanie fragmentów tekstu wejściowego pasujących do wzorca podanymi napisami: metody `replaceAll` i `replaceFirst`.

Odpowiednie są i w klasie `String`.

```
def txt = 'text  with extra  spaces within'
println txt
txt = txt.replaceAll(' +', ' ')
txt = txt.replaceFirst('with', 'with no')
println txt
```

Result:

```
text  with extra  spaces within
text with no extra spaces within
```



regex

Odwołania zwrotne

W napisie stanowiącym argument `replace...` - tekst zastępujący - możemy odwoływać się do zawartości grup wzorca. Wtedy tekst zastępujący będzie zawierał zawartość grupy z wyrażenia.

```
def txt = 'Num 1 Number 200'
println txt
txt = txt.replaceFirst(/\w+\s+(\d+)\s+\w+\s+(\d+)/, 'Number [$2] Number ($1)')
println txt
```

Result:

Num 1 Number 200

Number [200] Number (1)

def txt = 'NrInd801212345PESEL1233'

println txt

```
txt = txt.replaceFirst(/([a-zA-Z]+?) (\d+?) ([a-zA-Z]+?) (\d+?) /,
                      '$3 $2 $1 $4')
```

println txt

PESEL 801212345 NrInd 1233

Odwołania do zawartości grup



Wykonanie kodu przy zastępowaniu

Groovy pozwala dostarczyć domknięcia w `replaceAll` i `replaceFirst`. Kod domknięcia jest wykonywany dla każdego dopasowania, a wynik domknięcia zastępuje fragment dopasowany do wzorca.

```
def txt = 'krzesło 50 PLN komplet 2000 PLN'
println txt
txt = txt.replaceAll(/(\d+) PLN/) { all, kwota->
    String.format("%.2f", kwota.toBigDecimal()/4.12) + ' EUR'
}
println txt
```

kolejne dopasowanie



Pierwsza grupa w dopasowaniu

Wynik na konsoli:

```
krzesło 50 PLN komplet 2000 PLN
krzesło 12,14 EUR komplet 485,44 EUR
```

Rozbiór

Metoda split - separatory to fragmenty pasujące do wzorca.

```
def text = '20 monday 100 tuesday 200 friday 500 sat'  
def list = text.split(/\D/)  
println list  
list = text.split(/\D+/)  
println list
```

Uwaga!

Result:

```
[20, , , , , , , , 100, , , , , , , , , 200, , , , , , , , , 500]  
[20, 100, 200, 500]
```

i jeszcze:

```
text = ' 20 monday 100 tuesday 200 friday 500 sat'  
list = text.split(/\D+/)  
println list
```

Result:

```
[, 20, 100, 200, 500]
```

Uwaga: należy uważnie przeczytać opis działania metody split w JDK API

Ułatwienia w klasie String (Groovy)

String find(String regex)

Finds the first occurrence of a regular expression String within a String.

String find(Pattern pattern)

Finds the first occurrence of a compiled regular expression Pattern

String find(String regex, Closure closure)

Finds the first occurrence of a regular expression String within a String.

String find(Pattern pattern, Closure closure)

Finds the first occurrence of a compiled regular expression Pattern

List findAll(String regex)

Finds all occurrences of a regular expression string within a String.

List findAll(Pattern pattern)

Finds all occurrences of a regular expression Pattern within a String.

List findAll(String regex, Closure closure)

Finds all occurrences of a regular expression string within a String.

[Zobacz dokumentację metod w Groovy JDK \(GDK\).](#)

Przykłady find i findAll

```
// znajdź pierwszy ciąg cyfr w tekście  
println 'xyz'.find(/d+/)  
println 'abcdefgh 100 defgj'.find(/\\d+/)
```

```
// znajdź wszystkie ciągi cyfr w tekście  
def list = 'a 1 b 2 c9 x110'.findAll(/\\d+/)  
println list
```

Wynik:

```
null      // null znaczy, że nie znaleziono  
100  
[1, 2, 9, 110]
```

Metody find... z domknięciem

Do domknięcia przekazywane jest całe dopasowanie, a jeśli w wyr. regularnym występują grupy, to - jako kolejne argumenty- dopasowania do grup.

Kod domknięcia przetwarza uzyskane argumenty i zwraca wynik przetwarzania.

```
def res = 'abcdefgh 100 defgj'.find(/\d+/) { match->
    match.toInteger() + 11
}
println res
res = 'ble kwota 100 PLN ble'.find(/(kwota)\s+(\d+)\s+PLN/) {
    match, word, num ->
    'amount ' + num.toBigDecimal()*4.1 + ' EUR'
}
println res
```

Wynik:

111

amount 410.0 EUR

Przykład findAll z domknięciem

```
def list = '''
  1111 Jan
  2222 Ania
  3333 Stefan'''

list = list.findAll(/(?s)(\d\d\d\d)\s+(\w+)/) { all, nr, name ->
  "$nr@pjawstk.edu.pl <$name>"
}

list.each { println it }
```

Wynik:

```
s1111@pjawstk.edu.pl <Jan>
s2222@pjawstk.edu.pl <Ania>
s3333@pjawstk.edu.pl <Stefan>
```


Pattern jako klasyfikator w switch

```
def list = [1, -1, 100, '7', 'a', 2, 'x', 1.3 ]
def posInt = []
def other = []
list.each { e->
  switch(e) {
    case ~/\d+/ : posInt << e; break
    default: other << e
  }
}
println posInt
println other
```

Obiekt typu Pattern



Output:

```
[1, 100, 7, 2]
[-1, a, x, 1.3]
```

Przy napotkaniu case z klasyfikatorem typu Pattern, gdy element-kandydat nie jest napisem, to zostaje przekształcony w napis za pomocą metody toString() z jego klasy