

Kolekcje i obiekty iterowalne

© Krzysztof Barteczko, PJATK 2012-2017

Pojęcie kolekcji

Kolekcja jest obiektem, który grupuje elementy danych (inne obiekty) i pozwala traktować je jak jeden zestaw danych, umożliwiając jednocześnie wykonywanie operacji na zestawie danych np. dodawania i usuwania oraz przeglądania elementów zestawu.

Mamy do dyspozycji m.in. następujące rodzaje zestawów danych:

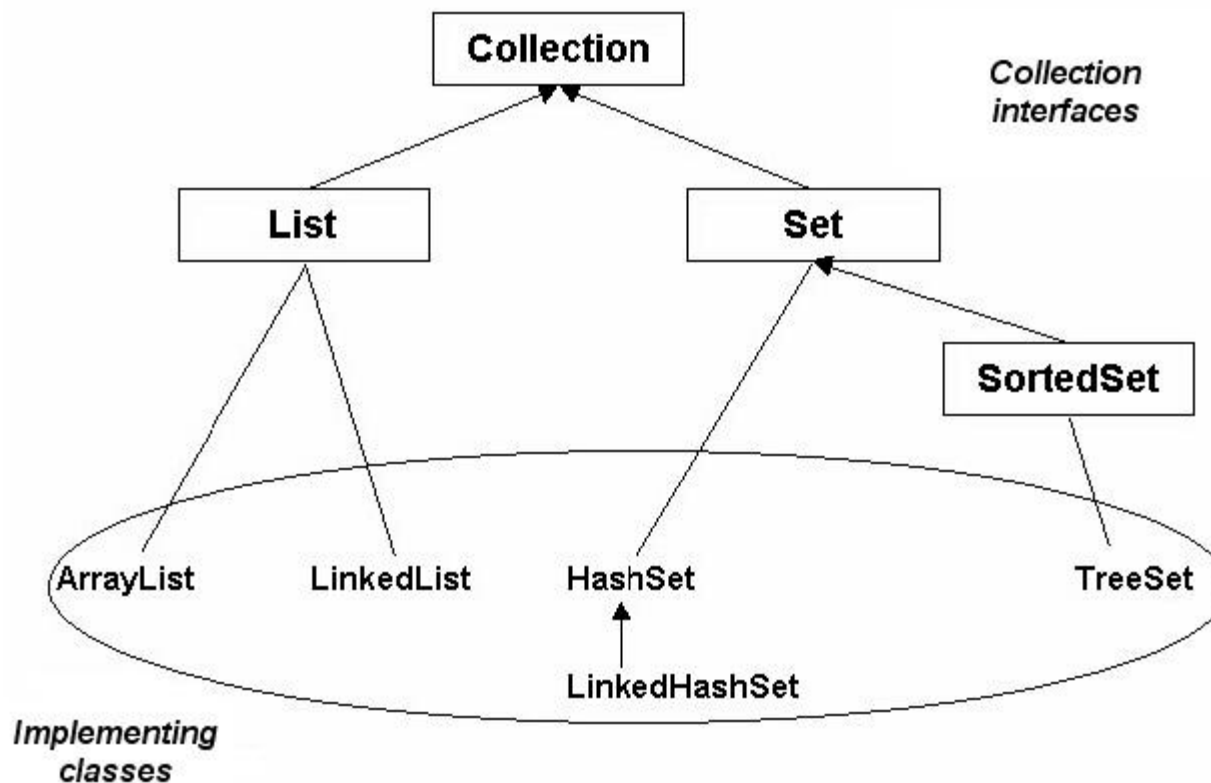
- listy (elementy mają pozycje i mogą się powtarzać)
- kolejki
- zbiory (elementy nie mają pozycji i nie mogą się powtarzać)
- mapy (zestaw odwzorowań klucz -> wartość)

Abstrakcyjne (niezależne od sposobu realizacji) właściwości tych struktur danych opisywane są przez interfejsy, a ich konkretne realizacje przez implementacje interfejsów w klasach.

Java Collections Framework:

interfejsy, implementacje (klasy), algorytmy

Podstawowe interfejsy i klasy kolekcyjne



[Zobacz bardziej szczegółowy opis](#)

Zalety stosowania kolekcji

```
def list = [] // ArrayList by default
def set1 = [] as HashSet // unsorted set
def set2 = [] as TreeSet // sorted set
def set3 = [] as Set // LinkedHashSet (preserve insert order)

// w każdym wierszu pliku - nazwa firmy, nie wiemy ile ich jest
// i czy nazwy się nie powtarzają
def lines = new File('firms.txt').readLines()
println lines
lines.each {
    list << it.trim()
    set1 << it.trim()
    set2 << it.trim()
    set3 << it.trim()
}
println list
println set1
println set2
println set3

// Wynik
[IBM , Sun, Oracle, Google, Sun, Apple]
[IBM, Sun, Oracle, Google, Sun, Apple]
[Google, Apple, IBM, Oracle, Sun]
[Apple, Google, IBM, Oracle, Sun]
[IBM, Sun, Oracle, Google, Apple]
```

Łatwość:

- dynamiczne tablice
- usuwanie duplikatów
- porządek

Kolekcje z kolekcji

Z każdej kolekcji można łatwo uzyskać inną (innego rodzaju):

```
def lines = new File('firms.txt').readLines()
println lines
def set1 = new HashSet(lines)
def set2 = new LinkedHashSet(lines)
println set1
println set2
// Można również tak
set2 = lines as Set // to będzie LinkedHashSet
def set3 = set1 as TreeSet
println set2
println set3
```

Wynik:

```
[IBM , Sun, Oracle, Google, Sun, Apple]
[Google, Apple, Oracle, Sun, IBM ]
[IBM , Sun, Oracle, Google, Apple]
[IBM , Sun, Oracle, Google, Apple]
[Apple, Google, IBM , Oracle, Sun]
```

Ogólne operacje na wszystkich kolekcjach

Najprostsze ogólne operacje na kolekcjach:

Metoda

Groovy operator

size()

isEmpty()

ref in (cond) np. if (coll) ...

add(Object)

<< +=

remove(Object)

clear()

contains(Object)

+ // tworzy nową kolekcję jako sumę

- // tworzy nową kolekcję jako różnicę

*n // duplikowanie

[] // dostęp do elementów po indeksach

Ogólne operacje na kolekcjach - przykład

```
def col // dowolna kolekcja
// Domknięcie testowe
def op = { closure ->
  col = [1,2,3] // za każdym razem inicjujemy na nowo
  closure()     // wykonanie operacji
  println col   // jak teraz wygląda
  col           // zwracamy ją do ew. wykorzystania
}
op { col << 100 }
op { println col + 100 } // tworzy nową nie zmieniając oryginału
op { col += 100 }
op { col << [ 10,11, 12] }
op { println col*2 } // tworzy nową nie zmieniając oryginału
op { col *= 2 }
def newcol = op { col << 7 << 8 << 9 }
println newcol
def diff = newcol - [1, 7]
println diff
```

Wynik

```
[1, 2, 3, 100]
[1, 2, 3, 100]
[1, 2, 3]
[1, 2, 3, 100]
[1, 2, 3, [10, 11, 12]]
[1, 2, 3, 1, 2, 3]
[1, 2, 3]
[1, 2, 3, 1, 2, 3]
[1, 2, 3, 7, 8, 9]
[1, 2, 3, 7, 8, 9]
[2, 3, 8, 9]
```

Iterator

Iterator jest obiektem klasy implementującej interfejs `Iterator` i służy do przeglądania elementów kolekcji oraz ew. usuwania ich przy przeglądaniu

Od każdej kolekcji możemy uzyskać iterator:

```
Iterator iter = c.iterator()
```

gdzie: `c` - dowolna klasa implementująca interfejs `Collection`.

Metody:

`hasNext()` - czy jest następny element

`next()` - daj następny element

`remove()` - usuń element zwrócony przez ostatnie `next`

Iterator - przykład

```
def set = [1, 2, 8, 8, 17, 9] as Set
println set
def iter = set.iterator()
while (iter.hasNext()) {
    elt = iter.next()
    println elt
    if (elt < 7) iter.remove()
}
println set
```

Output:

```
[17, 1, 2, 8, 9]
17
1
2
8
9
[17, 8, 9]
```

"Concurrent modification" niedozwolone

W trakcie iteracji za pomocą iteratora nie wolno modyfikować kolekcji innymi sposobami niż użycie metod tego iteratora (np. `remove()`).

Niech `c` - dowolna kolekcja

```
Iterator it1 = c.iterator();
while (it1.hasNext()) {
    it1.next()
    l.add('x') // Błąd fazy wykonania
}
```

```
Iterator it1 = c.iterator();
Iterator it2 = c.iterator();
while (it1.hasNext()) {
    it1.next();
    it2.next();
    it2.remove(); // Błąd fazy wykonania
}
```

ConcurrentModificationException



Iterowanie po dowolnych kolekcjach

Znane nam formy:

for (*var* in *col*) *ins*

for (*Typ var* : *col*) *ins*

col.each { closure }

col.forEachWithIndex { e, i -> ... }

Przykład:

```
def set = new HashSet()
def list = []
```

```
for (e in ['a','b','c', 'c']) set << e
set.each { list << it + 1 }
println set
println list
```

Output:

```
[b, c, a]
[b1, c1, a1]
```

Pod spodem są iteratory więc w kodzie nie można strukturalnie modyfikować kolekcji.

```
list.each {
  if (it[0] == 'a') list.remove(it)
}
```

ConcurrentModificationException

Obiekty iterowalne (1)

Za pomocą w/w metod iterować można po dowolnych obiektach klas implementujących Iterable.

Dodatkowo Groovy zapewnia iterowalność pewnych klas np. File czy String (dynamicznie dodając do nich iteratory w trakcie wywołania metod takich jak each(..)). Przykład:

```
// Różne formy iterowania
```

```
def iter = {  
    it.each { print "$it " }  
    println()  
    it.eachWithIndex { e, i -> print "$i - $e " }  
    println()  
    for (e in it) print "$e "  
    println()  
    for (def e : it) print "$e "  
    println()  
}}
```

```
iter new File('firms.txt') // dla plików tekstowych - po wierszach  
iter 'abcd' // dla napisów - po znakach  
iter (1..7) // dla zakresów - po elementach
```

Obiekty iterowalne (2)

Wynik poprzedniego programu:

```
IBM Sun Oracle Google Sun Apple
0 - IBM 1 - Sun 2 - Oracle 3 - Google 4 - Sun 5 - Apple
IBM Sun Oracle Google Sun Apple
IBM Sun Oracle Google Sun Apple
a b c d
0 - a 1 - b 2 - c 3 - d
a b c d
a b c d
1 2 3 4 5 6 7
0 - 1 1 - 2 2 - 3 3 - 4 4 - 5 5 - 6 6 - 7
1 2 3 4 5 6 7
1 2 3 4 5 6 7
```

Obiekty iterowalne (3)

Latwo też można tworzyć własne obiekty iterowalne. Przykład: dni miesiąca w tym tygodniu:

```
class CurrentWeek implements Iterable {  
    def start= new Date(),  
        end = start + 6,  
        curr = start  
  
    Iterator iterator() {  
        [ hasNext: { curr <= end },  
          next: { (curr++).format('dd') }  
        ] as Iterator  
    }  
  
    def reset() { curr = start }  
}
```

Sposób implementacji interfejsów.
Gdy interfejs ma więcej niż 1 metodę abstrakcyjną – mapa z kluczami = nazwy metod, wartościami = kody jako domknięcia. Przy jednej metodzie interfejsu wystarczy samo domknięcie. Mapę rzutujemy na nazwę interfejsu (tu **as Iterator**)

Po wyczerpaniu elementów iterator nie będzie działał (jest ustawiony za ostatnim elementem) Podana metoda reset() pozwala na ponowne użycie iteratora.

Obiekty iterowalne (4)

Użycie klasy CurrentWeek:


```
def cweek = new CurrentWeek()

cweek.each { print "$it " }
println()
cweek.reset()
cweek.eachWithIndex { e, i -> print "$i - $e " }
println()
cweek.reset()
for (d in cweek) print "$d "
println()
cweek.reset()
for (def d : cweek) print "$d "
```

Wynik:

```
11 12 13 14 15 16 17
0 - 11 1 - 12 2 - 13 3 - 14 4 - 15 5 - 16 6 - 17
11 12 13 14 15 16 17
11 12 13 14 15 16 17
```

Więcej na temat iteratorów



[Zobacz więcej
na temat iteratorów](#)

Dostęp do kolekcji po indeksach

W przeciwieństwie do Javy, Groovy dla każdego rodzaju kolekcji (a mówiąc ogólniej – dla każdego Iterable) zapewnia dostęp do elementów po ich indeksach.

```
def byIndx = { obj, i->
    println ''+obj.getClass() + " - indeks $i - " + obj[i]
}
```

```
def cols = [
    ['b', 'a', 'c', 'd'] as HashSet,
    ['b', 'a', 'c', 'd'] as Set,
    ['b', 'a', 'c', 'd'] as TreeSet
]
```

```
cols.each { byIndx it, 1 }
println new CurrentWeek()[2]
```

wynik:

```
class java.util.HashSet - indeks 1 - b
class java.util.LinkedHashSet - indeks 1 - a
class java.util.TreeSet - indeks 1 - b
13
```

Dla niektórych rodzajów kolekcji nie ma to wielkiego sensu, ale np. w przypadku LinkedHashSet czy TreeSet czasem chcielibyśmy łatwo uzyskać i-ty element w kolejności.

Uwaga: tylko dla ArrayList dostęp taki jest efektywny.

Dostęp do kolekcji po indeksach - ograniczenia

Tylko w przypadku list będziemy mogli ustalać wartości elementów "pod indeksami".

```
list[3] = 'abc' // Ok
```

ale nie:

```
set[3] = 'xxx' // błąd
```

Niuanse usuwania elementów (1)

Metoda `remove(obiekt)` usuwa z dowolnej kolekcji pierwszy element, który spełnia warunek `elt.equals(obiekt)`.

Natomiast operator `"-"` usuwa wszystkie takie elementy:

```
def col = [ 'a', 'b', 'a', 'c' ]  
col.remove('a')  
println col           // modyfikuje oryginał  
col = [ 'a', 'b', 'a', 'c' ]  
println col-'a'       // zwraca nową kolekcję  
println col           // nie modyfikuje oryginału
```

Wynik:

```
[b, a, c]  
[b, c]  
[a, b, a, c]
```

Niuanse usuwaniu elementów (2)

Dla kolekcji listowych metoda `remove(liczba_całkowita)` z JDK usuwa element pod podanym indeksem i zwraca usunięty element:

```
col = [2, 3, 1]
println col.remove(1) // 3
println col           // [2, 1]
```

A jak usunąć z tej listy jedynkę?

Groovy: `col.removeElement(1)`

Java: `col.remove(Integer.valueOf(1))` // w Groovy to nie działa

Dla innych rodzajów kolekcji jest tylko `remove(Object)`, więc nie ma usuwania po indeksach ani dwuznaczności.

Operacje zbiorowe

Oprócz grupowych operacji na kolekcjach z JDK, Groovy udostępnia nowe m.in.



[Zobacz](#)

dodawanie elementów tablicy - **addAll (Object[])**

dodawanie elementów dowolnego Iterable – **addAll(Iterator)**

usuwanie elementów zawartych w tablicy –
removeAll(Object[])

usuwanie/pozostawianie elementów spełniających jakiś warunek: **removeAll(Closure)**, **retainAll(Closure)**

a także użyteczne operacje (działające ogólnie na Iterable):

intersect – część wspólna (suma jest zapewniana przez +),

unique (dla Iterable - **toUnique**) – usuwanie duplikatów,

toSet, **toList** itp.,

flatten – spłaszczanie kolekcji.

Warunkowe usuwanie - przykład

Z kolekcji miejsc usuniemy wszystkie nie-wyspy:

```
def col = [ "Cypr - wyspa",  
            "Madagaskar - wyspa",  
            "Krabi",  
            "Londyn"]
```

```
col.removeAll { !it.contains('wyspa') }  
// Można i tak  
col.retainAll { it.endsWith('wyspa') }  
// Albo też korzystając z  
// metody removeIf z Javy 8  
col.removeIf { !it.endsWith('wyspa') }
```

Tu widać, że wszędzie tam, gdzie w Javie wymagane jest lambda-wyrażenie – można podać domknięcie.

Uwaga: wszystkie w/w metody modyfikują oryginalną kolekcję. Aby uzyskać nową kolekcję z usuniętymi elementami, nie zmieniając oryginału, można zastosować metodę `findAll()`.

Spłaszczanie kolekcji

Użyteczna metoda `flatten()`, mające ogólne zastosowanie do dowolnego Iterable, spłaszcza zestaw danych (mogący zawierać jako elementy inne zestawy). Przykład:

```
def col1 = [1, 2, 3]
def col2 = [7, 8, 9]
col2 << [10, 11, 12]
col1 << col2
println col1
col1 = col1.flatten()
println col1
```

wynik:

```
[1, 2, 3, [7, 8, 9, [10, 11, 12]]]
[1, 2, 3, 7, 8, 9, 10, 11, 12]
```

Filter-map-reduce

Na każdym obiekcie iterowalnym można stosować operacje typu filter-map-reduce

(zob. wprowadzenie do tego tematu w Javie)

W języku Groovy mamy m.in. następujące metody:

filter === findAll

map === collect

reduce === inject (+ różne proste redukcje
+ redukcje kontenerowe)

Metoda collect

List list = coll.collect { closure }

Iterates through this collection transforming each entry into a new value using the closure as a transformer, returning a list of transformed values.

```
def defaultBonus = 100
def f = new File('bonus.txt')
println f.text
def bonus = f.readlines().collect {
  if (it.tokenize().size() < 2) "$it $defaultBonus"
  else it
}
println 'Default bonus assigned'
bonus.each { println it }
```

Output:

```
John 700
Steve
Kate 500
Adam
Default bonus assigned
John 700
Steve 100
Kate 500
Adam 100
```

collectEntries

collectEntries

public Map **collectEntries**(Closure transform)

Iterates through this Collection transforming each item using the transform closure and returning a map of the resulting transformed entries.

```
def letters = "abc"
// collect letters with index using list style
def n = letters.size()-1
def map = (0..n).collectEntries { ind -> [ind, letters[ind]]}
println map

// collect letters with index using map style
map = (0..n).collectEntries { ind -> [(ind) : letters[ind]] }
println map

def l1 = [ 'a', 'b', 'c' ]
def l2 = [1,2,3]
map = (0..<l1.size()).collectEntries { i -> [ l1[i] , l2[i]] }
println map
```

wynik:

```
[0:a, 1:b, 2:c]
[0:a, 1:b, 2:c]
[a:1, b:2, c:3]
```

Filtrowanie

M.in. metody:

Collection findAll(Closure closure)

Finds all values matching the closure condition.

Collection grep(Object filter)

Iterates over every element of the collection and returns each item that matches the given filter (filter as in 'case' label of 'switch' statement)

Inne metody:

`find { Closure }` // zwraca pierwszy element spełniający warunek

`boolean any { Closure }` // czy jest elt spełniający warunek

`boolean every { Closure }` // czy wszystkie spełniają warunek

Przykład

```
def allFiles = []
def dir = new File('..')
dir.eachFileRecurse { if (it.isFile()) allFiles << it }
def groovyFiles = allFiles.findAll {
    it.name.endsWith('.groovy')
}
def smallFiles = allFiles.grep( { it.size() < 1000 } )
def smallFileNames = smallFiles.collect { it.name }
def txtFileNames = smallFileNames.grep( ~/\.+\.txt/)

println 'Files in dirs of ' + dir.canonicalPath
println 'Is there any smaller than 100 ' + allFiles.any { it.size() < 100 }
println 'Is every file older than now ' +
    allFiles.every { it.lastModified() < new Date().time }
println 'All: ' + allFiles.size()
println 'Small: ' + smallFiles.size()
println 'Groovy: ' + groovyFiles.size()
println 'Small text files: ' + txtFileNames.size()
```

Output (could be):

```
Files in dirs of E:\PJWSTK\InfSpol\Wyklady\IspWprWsp
Is there any smaller than 100 true
Is every file older than now true
All: 1121
Small: 880
Groovy: 89
Small text files: 27
```

Metoda withIndex

Dostępna od wersji 2.4 pozwala na użycie indeksów w takich operacjach jak collect czy findAll (zob. dokumentację w GDK).

Przykład:

```
def col = [ 100, 200, 300, 400, 500, 600 ]

// withIndex daje listę par [element, indeks]
def wi = col.withIndex()
println wi.getClass()
println wi
// inaczej transformujemy elementy o przystych-nieprzystych indeksach
println col.withIndex().collect { e, i -> i % 2 ? e/100 : e/10 }
// findAll będzie zwracać pary [element, indeks] spełniające podany warunek
println col.withIndex().findAll { e, i -> e > 300 }
println col.withIndex().findAll { e, i -> i > 2 }
println col.withIndex().findAll { e, i -> i < 2 ? e > 100 : e > 500 }
```

Wynik:

```
class java.util.ArrayList
[[100, 0], [200, 1], [300, 2], [400, 3], [500, 4], [600, 5]]
[10, 2, 30, 4, 50, 6]
[[400, 3], [500, 4], [600, 5]]
[[400, 3], [500, 4], [600, 5]]
[[200, 1], [600, 5]]
```

Metoda inject

Z dokumentacji GDK:

Object inject(Object value, Closure closure)

Iterates through the given object, passing in the initial value to the closure along with the current iterated item then passing into the next iteration the value of the previous closure.

```
def taskNr = 2
def name = "Kowalski Jan Maria"
def nameWords = name.tokenize(' ')
def proj = nameWords.inject("Task$taskNr") { p, elt -> p +=elt[0] }
println proj
```

Output:
Task2KJM

Sumowanie

Object sum()

Sums the items in a collection.

Z dokumentacji GDK

Object sum(Object initialValue)

Sums the items in a collection, adding the result to some initial value.

Object sum(Closure closure)

Sums the result of apply a closure to each item of a collection.

Object sum(Object initialValue, Closure closure)

Sums the result of apply a closure to each item of a collection to sum initial value.

```
s1 =[1, 5, 7].sum()
s2 = [1, 5, 7].sum(10)
s3 = new File('bonus.txt').readLines().sum { it.size() }
println "$s1 $s2 $s3"
```

Output:

```
13 23 26
```

Operator spread i spread-dot

Spread === a(*list) a(list[0], list[1], ...)

Spread-dot === list*.a() [list[0].a(), list[1].a() ...]

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

```
def inp = showInputDialog('Enter 3 numbers')
```

```
println inp
```

```
if (inp) {
```

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

```
    println list.sum()
```

```
    def nums = list*.toInteger()
```

```
    println nums.sum()
```

```
    println oper(*nums)
```

```
}
```

```
def oper (a, b, c) {
```

```
    return a * (b + c)
```

```
}
```

Output:

1 2 3

123

6

5

Więcej metod ...

GDK zawiera bardzo dużo metod użytecznych do pracy z kolekcjami i obiektami iterowalnymi. Do ciekawych należą np. metody uzyskiwania kombinacji i permutacji elementów oraz iterowania po nich, a także metody indeksowania (zipowania z indeksami) dowolnego Iterable.

Zobacz dokumentację GDK:

Iterable -> <http://users.pja.edu.pl/~kb/PJPadd/docs/Iterable.html>

Collection -> <http://users.pja.edu.pl/~kb/PJPadd/docs/Collection.html>