

# Listy, zbiory, tablice

© Krzysztof Barteczko, PJATK 2012-2017

# Listy

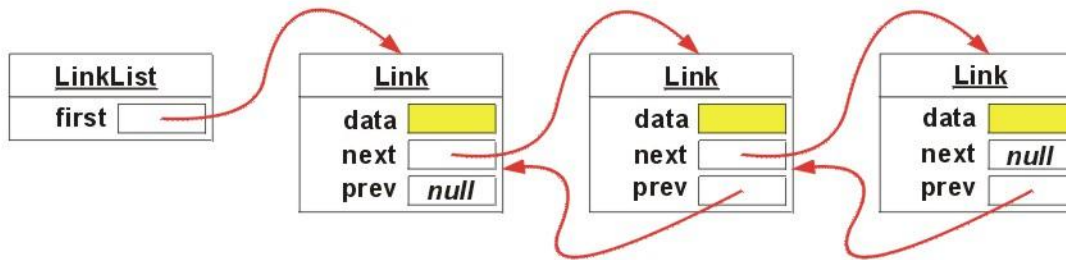
Implementowane interfejsy: List

Implementacje:

ArrayList - dynamiczna tablica  
(szybki dostęp bezpośredni)

LinkedList - lista liniowa z podwójnymi dowiązaniem  
(szybkie usuwanie i dodawanie w środku)

[Zob. więcej na temat operacji na listach w JDK](#)




Z tego wynikają istotne różnice w efektywności operacji dodawania elementów. Przedstawiony dalej test efektywności ma oczywiście przybliżony charakter (nie uwzględnia kwestii związanych z rozgrzewką JVM oraz odśmiecaniem), ale dobrze pokazuje różnice.

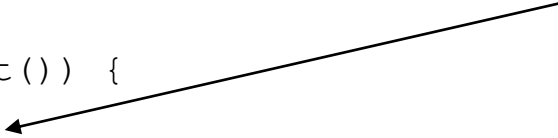
# Test efektywności - funkcje

```
def randomAccess(List list) {  
  Random rand = new Random();  
  10000.times {  
    def index = rand.nextInt(list.size())  
    def s = list[index]  
    list[index] = s + 'a'  
  }  
}
```

Generator liczb  
pseudolosowych



```
void insert(List l) {  
  ListIterator iter = l.listIterator() // has additional methods  
  int i = 0  
  while (iter.hasNext()) {  
    iter.next();  
    if (i % 2) iter.add('b');  
    i++;  
  }  
}
```



# Efektywność test

```
long start;

def setTimer = { start = System.currentTimeMillis(); }
def elapsed = { return System.currentTimeMillis() - start; }

ArrayList aList = []
100000.times() { aList << 'a' }
LinkedList lList = new LinkedList(aList)
setTimer()
randomAccess(aList)
println "Direct access to ArrayList: " + elapsed() + " ms"
setTimer();
randomAccess(lList);
println "Direct access to LinkedList: " + elapsed() + " ms"
setTimer();
insert(aList);
System.out.println "Inserting in ArrayList: " + elapsed() + " ms"
setTimer();
insert(lList);
System.out.println "Inserting in LinkedList: " + elapsed() + " ms"
```

## Output

```
Direct access to ArrayList: 109 ms
Direct access to LinkedList: 9969 ms
Inserting in ArrayList: 8594 ms
Inserting in LinkedList: 297 ms
```

# Tworzenie list

```
// Bezposrednia inicjacja
def list = [1, 2, 3] // domyslnie ArrayList
// Bezposrednia inicjacja jako LinkedList
list = ['a', 'b', 'c'] as LinkedList

// Z tablicy
String[] stab = 'ala ma kota'.split()
println stab.toList()

// Z dowolnej kolekcji
def set = ['x', 'y', 'z'] as Set
println set.toList()

// Z napisu
println 'abcdefg'.toList()

// Z dowolnego Iteratora
def start = 1, end = 3, curr = 0
def niter = [hasNext: { curr < end }, next: { ++curr } ] as Iterator
println niter.toList()

// Z dowolnego Iterable
curr = 0
def numitb = { niter } as Iterable
println numitb.toList()
```

Wynik:

[ala, ma, kota]

[x, y, z]

[a, b, c, d, e, f, g]

[1, 2, 3]

[1, 2, 3]

# Dostęp do elementów po indeksach

Inaczej niż Java, Groovy pozwala odwoływać się do (prawie) dowolnych wartości indeksów:

```
def list = ['a', 'b']  
println list[100] // wartość null a nie wyjątek ArrayIndexOutOfBoundsException
```

Jak wiemy, ujemne indeksy oznaczają "odliczanie" od końca:

list[-1] to 'b', list[-2] to 'a',

natomiast podanie zbyt dużego ujemnego indeksu (wykraczającego poza początek listy) spowoduje powstanie wyjątku.

## Listy – wartości domyślne (1)

Możliwość sięgania po dowolne indeksy implikuje też możliwość ustalenia domyślnej wartości elementu, gdy indeks wykracza poza rozmiar listy.

```
def list = [90, 85].withDefault { 100 }  
println list[3] // nieistniejący element uzyskał wart. 100  
println list // ale lista zwiększyła swoje rozmiary:
```

wynik:

100

[90, 85, null, 100]

Zauważmy, że "niedotknięty" element ma wartość null. Jest to tzw. leniwe wypełnianie – wartość domyślna została nadana tylko jednemu elementowi.

## Listy – wartości domyślne (2)

Parametrem metody `withDefault` jest domknięcie. Dostaje ono jako argument indeks i kod domknięcia i może na tej podstawie inicjować nieistniejące elementy:

```
def letters = [].withDefault { i-> 'A'*i }  
println letters[5]  
println letters
```

wynik:

```
AAAAA  
[null, null, null, null, null, AAAAA]
```

Leniwość inicjacji jest dobrym rozwiązaniem, jeśli inicjacja jest kosztowna. Ale czasem chcemy zamiast `null` mieć odpowiednie wartości. Do tego służy metoda `withEagerDefault`:

```
def list = [90, 85].withEagerDefault { 100 }  
println list[5]  
println list
```

wynik:

```
100  
[90, 85, 100, 100, 100, 100]
```



# Listy – użycie wielu indeksów

Jak wiemy, pobierać i ustalać elementy list można za pomocą operatora indeksowania: `x = list[0]`; `list[-1] = 111` (-1 oznacza ostatni element).

Podlisty można uzyskiwać używając zakresów indeksów. Zakresów można też używać przy nadawaniu wartości:

```
def list = ['a', 'b', 'c', 'xxxxx' ]

println list[1..2] // inaczej: list.subList (1, 3)
println list[1..<2] // wyłączaąco - inaczej: list.subList (1, 2)

// Odwracanie
println list[-1..0] // inaczej: list.reverse()

// Nadawanie wartości
list[1..2] = [ 'x', 'x' ]
println list

// Zamiast zakresów można stosować dowolne kolekcje
idx = [0, 3]
println list[idx]
list[idx] = ['x']*idx.size()
println list
```

Wynik:

```
[b, c]
[b]
[xxxxx, c, b, a]
[a, x, x, xxxxx]
[a, xxxxx]
[x, x, x, x]
```

# Użyteczne operacje na listach

Groovy dostarcza wielu użytecznych metod do operowania na listach. Są to oczywiście wszystkie metody GDK z klasy `Object`, `Iterable` i `Collection`

Dla list mamy dodatkowo:

`transpose()` – transpozycja

`reverseEach(Closure)` – `each` w kolejności odwrotnej

`removeAt(index)` – usuwa i zwraca element pod indeksem

`pop()` – usuwa i zwraca ostatni element listy

Biorąc pod uwagę uporządkowany (wg pozycji) charakter list do szczególnie użytecznych należą metody `drop/take` (usuwania – pobierania elementów z końca/początku) – również w wersjach z domknięciem jako selektorem elementów (usuwaj/pobieraj dopóki elementy spełniają warunek) oraz operacji odnajdywania indeksów elementów spełniających warunek podany w domknięciu.

# Operacje na listach – przykład transpose()

W pliku TSV mamy dane o produkcji, zatrudnieniu i kapitale wg lat (kolumny to lata, wiersze – kategorie).

Obróbka statystyczna wymaga innego porządku: lata – w wierszach, kategorie w kolumnach. Użyjemy transpozycji:

```
def data = []  
new File('data.tsv').splitEachLine('\t') {  
    data << it  
}  
data.each { println it.join('\t') }  
println 'Transpozycja'  
data = data.transpose()  
data.each { println it.join('\t') }
```

Wynik:

	2015	2016	2017
Prod	100	110	120
Empl	40	42	43
Assets	1000	1100	1150
Transpozycja			
	Prod	Empl	Assets
2015	100	40	1000
2016	110	42	1100
2017	120	43	1150

# Operacje na listach – wyszukiwanie indeksów

Przykłady:

```
def list = ['x', 'aaa', 'abc', 'y']

println list.findIndexOf { it.startsWith('a') }
println list.findIndexValues { it.startsWith('a') }
println list.findIndexOf { it.size() == 1 }
println list.findIndexValues { it.size() == 1 }

list = [ 100, 1, 7, 21, 100, 51 ]
println list.findIndexValues { it in (20..100) }
```

Wynik:

```
1
[1, 2]
0
[0, 3]
[0, 3, 4, 5]
```

# Wyszukiwanie indeksów - przykład praktyczny

W pliku TSV znajdują się dane o jakimś zjawisku (biznesowym, meteorologicznym) dla szeregu lat od 2000 roku. Np.

2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015
21	15	13	15	80	12	10	26	17	30	72	38	49	41	35	73
74	7	46	10	69	2	40	23	11	73	18	28	95	5	57	5
30	43	49	82	91	99	64	13	47	38	20	6	33	16	26	48

Należy utworzyć z nich tablicę danych (TSV) dla wybranych lat (np. co pięć lat).  
Rozwiązanie:

```
data = new File('data2.tsv').collect { it.tokenize('\t') }
years = data.remove(0)
selYearsInd = years*.toInteger().findIndexValues { it%5==0 }
```

```
data = data.collect { it[selYearsInd] }
out = [ years[selYearsInd].join('\t') ]
data.each { out << it.join('\t') }
println out.join('\n')
```

Wynik:

2000	2005	2010	2015
21	12	72	73
74	2	18	5
30	99	20	48

```
// Prostsze rozwiązanie
// ale wymagające dwukrotnej transpozycji
data = new File('data2.tsv')
    .collect { it.tokenize('\t') }
    .transpose()
    .findAll { it[0].toInteger()%5 == 0 }
    .transpose()
    .collect { it.join('\t') }.join('\n')
println data
```

# Maksimum, minimum, sortowanie

Metody odnajdywania maksimum i minimum oraz sortowania określone są dla dowolnego Iterable (także dla Iterator). Rozważymy je na przykładzie list.

```
def list = [ 100, 1, 20, 40, 5]
println list.max()
println list.min()

// sortuje i zwraca oryginał (lista jest modyfikowana)
list1 = list.sort()
println list
println list.is(list1) // czy list i list1 wskazuje na ten sam obiekt

// z argumentem mutate == false
// tworzy nową, posortowaną listę, nie zmienia oryginału
list = [ 100, 1, 20, 40, 5]
list1 = list.sort(false)
println list
println list.is(list1) // czy list i list1 wskazuje na ten sam obiekt
```

Wynik:

```
100
1
[1, 5, 20, 40, 100]
true
[100, 1, 20, 40, 5]
false
```

# Porządek

Skąd wiadomo, że 100 jest większe od 1?

Otóż odnajdywanie ekstremów, sortowanie (a także dodawanie elementów do zbiorów uporządkowanych (np. TreeSet), iterowanie po kolekcjach uporządkowanych, pobieranie elementów z kolejek z priorytetami) odbywa się w oparciu o:

- \* naturalny porządek obiektów,
- \* lub reguły porównywania obiektów określone przez komparator - obiekt klasy implementującej interfejs Comparator.

W jakich klasach jak pochodne od Number czy klasy napisowe – określony jest porządek naturalny, dlatego w poprzednich przykładach wszystko działało bez problemów. We własnych klasach trzeba taki porządek określać albo w metodach (np. sortowania) podać komparatory.

# Porządek naturalny

Porządek naturalny określany jest przez definicję metody `compareTo` w klasie. Ma ona następujący nagłówek

```
public int compareTo(T otherObject)
```

gdzie `T` jest typem obiektu,

i zwraca:

- \* liczbę  $< 0$ , jeżeli ten obiekt (`this`) znajduje (w porządku) przed obiektem `otherObject`,

- \* liczbę  $> 0$ , jeżeli ten obiekt (`this`) znajduje (w porządku) po obiekcie `otherObject`,

- \*  $0$ , jeśli obiekty są takie same.



# Komparator

Komparator jest obiektem porównującym inne obiekty

Komparatory są realizowane jako obiekty klas implementujących interfejs `Comparator<T>`, gdzie `T` jest typem porównywanych obiektów

Interfejs ten zawiera ważną metodę:

```
int compare(T o1, T o2)
```

W Groovy interfejsy z metodą `compare` imlementujemy tak:

```
def comp = [ compare: { o1, o2 -> ... } ] as Comparator
```

Powstaje  
obiekt-komparator

Nazwa metody

Closure

# Max, min i sort z komparatorami

W metodach max, min i sort możemy jako argument podać stworzony wcześniej komparator albo po prostu domknięcie, które stanowi kod komparatora:

```
def list = [ "Red", 'Green', 'Blue' ]
println list.sort()
println list.sort { o1, o2 -> o1.size() - o2.size() }

// uwaga uzycie metody sort z komparatorem
// oznacza wywołanie takiej metody interfejsu List z JDK (nie GDK)
// i typ wyniku jest void czyli zwracany jest null
list = [ "Red", 'Green', 'Blue' ]
comp = { o1, o2 -> o1.size() - o2.size() } as Comparator
println list.sort(comp)
println list
// aby uzyskać nową listę i nie zmieniać oryginału użyj metody
// z GDK: List sort(immutable, Comparator comparator)
```

Wynik:

```
[Blue, Green, Red]
[Red, Blue, Green]
null
[Red, Blue, Green]
```

# Specjalne komparatory

Odwrócony porządek naturalny łatwo uzyskujemy przez `Comparator.reverseOrder()`:

```
list = [ 'Red', 'Blue', 'Green' ]
list.sort(Comparator.reverseOrder())
println list // [Red, Green, Blue]
```

Sortowanie według reguł danej strefy językowej wymaga użycia odpowiedniej instancji kolatora (pakiet `java.text`). Przykład:

```
import java.text.Collator
list = ["cedr", "ćwiek", "list", "łopata" ]
println list.sort() // bez kolatora
// Kolator dla języka polskiego
def comp = Collator.getInstance(new Locale('pl'))
// kolatory są komparatorami
println (comp instanceof Comparator)
list.sort(comp)
println list
```

Wynik:

```
[cedr, list, ćwiek, łopata]
true
[cedr, ćwiek, list, łopata]
```

# Zbiory

Niepowtarzające się elementy -> zasadnicza jest efektywność stwierdzania czy element jest czy nie w zbiorze.

Wyszukiwanie na listach - nieefektywne (metoda contains()).

Dla zbiorów dwa różne podejścia:

- tablice mieszania - klasa **HashSet**,
- binarne drzewo do szybkiego wyszukiwania - klasa TreeSet.

Uboczny efekt zastosowania TreeSet - uporządkowanie.

Klasa LinkedHashSet - elementy w tej kolejności w jakiej były dodawane.

# Tablice mieszania

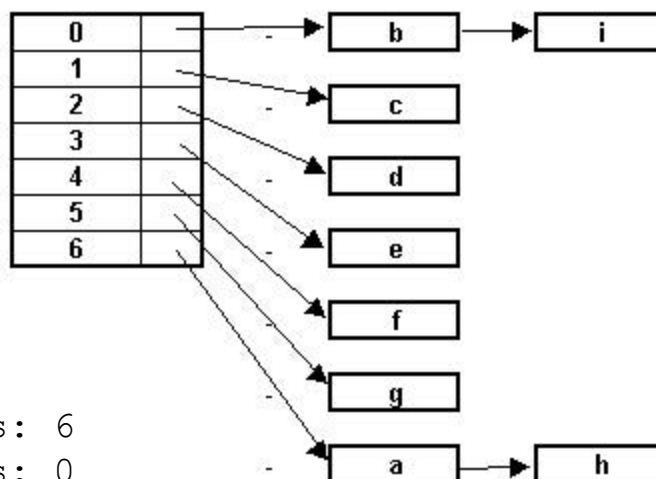
Tablica mieszania (**hashtable**) jest strukturą danych specjalnie przystosowaną do szybkiego odnajdywania elementów. Dla każdego elementu danych wyliczany jest kod numeryczny (liczba całkowita) nazywany kodem mieszania (hashcode), na podstawie którego obliczany jest indeks w tablicy, pod którym będzie umieszczony dany element. Może się zdarzyć, że kilka elementów otrzyma ten sam indeks, zatem elementy tablicy mieszającej stanowią listy (kubelki), na których znajdują się elementy danych o takim samym indeksie, wyliczonym na podstawie ich kodów mieszania.

W Javie można wyliczyć kod mieszania dla każdego obiektu za pomocą zastosowania metody `hashCode()`

Jeżeli prawdziwe jest `a.equals(b)`,  
to musi być spełniony warunek `a.hashCode() == b.hashCode()`.

# Hashtable przykład

Napisy "a", "b", "c", "d", "e", "f", "g", "h", "i" będą umieszczone w tablicy o 7 kubełkach



```
a kod: 97 indeks: 6
b kod: 98 indeks: 0
c kod: 99 indeks: 1
d kod: 100 indeks: 2
e kod: 101 indeks: 3
f kod: 102 indeks: 4
g kod: 103 indeks: 5
h kod: 104 indeks: 6
i kod: 105 indeks: 0
```

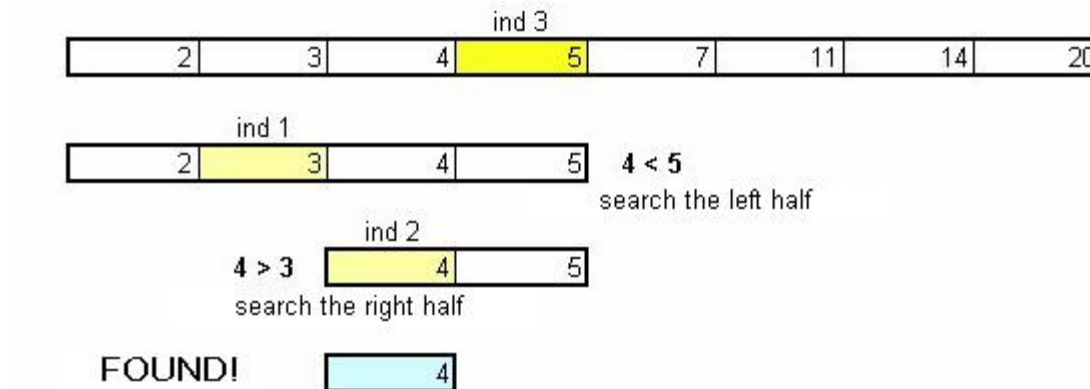
O tym czy element jest w HashSet decyduje metoda hashCode() oraz equals(Object) z klasy elementu.

# Binarne wyszukiwanie

Czy jest 4 w zestawie 3 4 2 5 7 20 11 14

Uporządkowane: 2 3 4 5 7 11 14 20

Teraz wyszukiwanie też jest szybkie. Tę ideę stosuje TreeSet.



# TreeSet – zawartość a porządek

O porządku decydują metody `compareTo` z klasy obiektów lub `compare` komparatora. W przypadku `TreeSet` decydują one nie tylko o porządku ale i o zawartości (elementy "takie same" w porządku jak już zawarty w zbiorze nie będą dodawane). Dlatego metody `compareTo` w klasach obiektów, a także komparatory używane w konstruktorze `TreeSet(Comparator)` muszą jednoznacznie rozróżniać elementy, które w zbiorze mają się znaleźć.

```
def list = ['ala', 'x', 'kot', 'pies' ]
def set1 = list as TreeSet // porządek naturalny
println set1

// porządek wg komparatora użytego w konstruktorze
def set2 = new TreeSet( {a, b -> a.size() - b.size() } )
set2.addAll(list)
println set2 // straciliśmy kota!

// jeżeli chcemy mieć posortowaną zawartość TreeSet
// wg dowolnego komparatora użyjemy metody sort(), która nie
// zmieni zawartości zbioru, a zwróci posortowaną listę
def list2 = set1.sort {a, b -> a.size() - b.size() }
println set1
println list2.getClass().simpleName
println list2
```

Wynik:

```
[ala, kot, pies, x]
[x, ala, pies]
[ala, kot, pies, x]
ArrayList
[x, ala, kot, pies]
```



Więcej o zbiorach ...



Zobacz więcej o zbiorach w JDK

# Tablice

**Tablice** są zestawami elementów (wartości) tego samego typu, ułożonych na określonych pozycjach.

Do każdego z tych elementów mamy bezpośredni ( swobodny - nie wymagający przeglądania innych elementów zestawu) dostęp poprzez nazwę tablicy i pozycję elementu w zestawie, określaną przez indeks lub indeksy tablicy.

Różnica pomiędzy tablicami a prostymi listami:

- > rozmiar tablicy niezmienny (po utworzeniu),
- > rozmiar listy może się zmieniać (dodawanie i usuwanie elementów)

Tablice - po co ?

- efektywność
- wiele metod z klas Javy zwraca tablice, lub wymaga arg. tablic
- niemodyfikowalna (co do rozmiaru) lista

# Definiowanie i tworzenie tablic

Tworzenie tablicy:  
new T[n]

gdzie:

\* T - typ elementów tablicy

\* n - rozmiar tablicy (liczba elementów tablicy)

Np.

```
arr = new BigDecimal[5]
```

```
def arr1
```

```
//...
```

```
arr1 = new Integer[3] // można też użyć typu int
```

```
// n jest typu Integer i ma wartość (np. wprowadzoną)
```

```
def arr2 = new String[n]
```

```
int[] a1
```

```
a1 = new int[10]
```

```
String[] stab = new String[n]
```

Tablica jest obiektem typu T[],

tu: int[] i String[]

Uwaga:

inicjacja tablic jak w Javie: int[] x = {1,2};

jest w Groovy niedopuszczalna. Używajmy:

int[] x = [1, 2]

# Dostęp do elementów tablic

indeksy - liczby całkowite typu int od 0

liczba elementów w tablicy:

`n = arr.length // Groovy i Java`

`n = arr.size() // Groovy`

dostęp do elementów: `arr[0], arr[1], ..., arr[n-1]`

np.:

```
def arr = new int[3]
```

```
arr[0] = 1
```

```
arr[1] = 2
```

```
arr[2] = 3
```

```
s = arr[1] + arr[2]
```

```
arr[0] = arr[0] * 2
```

# Tablice a proste listy - przykład

```
def arr = new String[2]
arr[0] = 'a'
arr[1] = 'b'
//arr << 'c'    błąd - operator << nie dla tablic
//arr[2]='c'    ArrayIndexOutOfBoundsException
println arr.class.name + ' ' + arr
println 'First: ' + arr[0]
println 'Last: ' + arr[arr.length-1]
println 'Last: ' + arr[arr.size()-1]
println 'Last: ' + arr[-1]

def list = []
list[0] = 'a'
list[1] = 'b'
list[2] = 'c'
println list.class.name + ' ' + list
println 'First: ' + list[0]
println 'Last: ' + list[list.size()-1]
//println 'Last: ' + list[list.length-1]    błąd length nie dla list
println 'Last: ' + list[-1]
println list
list << 'c'
println list
```

Output:

```
[Ljava.lang.String;
[a, b]
First: a
Last: b
Last: b
Last: b
java.util.ArrayList
[a, b, c]
First: a
Last: c
Last: c
[a, b, c]
[a, b, c, c]
```

## Konwersje lista <-> tablica

Tablice można przekształcać w listy i vice versa:

```
arr = [1, 2, 3] as Integer[] // można też as int[]
```

```
list = [ 'a', 'b', 'c' ]
```

```
arr = list as String[] // można też: arr = (String[]) list
```

```
list = arr as List // ten sam efekt co arr as ArrayList
```

```
list = arr.toList() // j.w.
```

# Tablice i listy - identyczne operowanie

Większość omówionych sposobów dostępu, iterowanie, przeszukiwania itp. może być używana jednolicie dla tablic i list.

```
def list = [1, 7, 15, 22 ]
def arr = [1, 2 , 11, 17] as int[]
println list.class.name + ' ' + arr.class.name
sameSyntax list
sameSyntax arr

def sameSyntax(obj) {
  obj.each { print it + ' ' }
  println()
  println obj.findAll { it > 10 }
  println obj.inject(0) { sum, elt -> sum += elt }
}
```

Output:

```
java.util.ArrayList [I
1 7 15 22
[15, 22]
45
1 2 11 17
[11, 17]
31
```

Zestaw dostępnych metod dla tablic określają klasy `Object` i `Object[]` z GDK - zob. dokumentację.