Mapy

Mapy

Mapa jest jednoznacznym odwzorowaniem zbioru kluczy w zbiór wartości.

kolekcja par: klucz - wartość, która zapewniają odnajdywanie wartości związanej z podanym kluczem.

Np. zestaw danych, który zawiera nazwiska i numery telefonów i pozwala na odnajdywanie numeru telefonu (poszukiwanej wartości) po nazwisku (kluczu).

Zarówno klucze, jak i wartości mogą być referencjami do dowolnych obiektów (jak również wartościami null).
Wartości kluczy nie mogą się powtarzać (odwzorowanie musi być jednoznaczne).
Natomiast pod różnymi kluczami można zapisać te same wartości (odzworowanie nie musi być wzajemnie jednoznaczne).

Użyteczność map (1)

Problem:
w wierszach pliku firmsAddr.tsv znajdują się (rozdzielone znakiem tabulacji) nazwy i adresy firm (w każdym wierszu jedna taka para). Nasz program po wczytaniu pliku ma za zadanie dostarczenie prostego i efektywnego interfejsu wyszukiwania adresu dla podanej nazwy firmy.
Rozwiązanie: mapa, klucze = nazwy, wartości = adresy.

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

def map = [:].withDefault{'n/a'}
new File('firmsAddr.tsv').eachLine {
    def (name, addr) = it.tokenize('\t')
    map[name] = addr
}

while (name = showInputDialog 'Enter firm name' ) {
  showMessageDialog null, "Firm: $name\nAddress: " + map[name]
}
```

Szybkie wyszukiwanie po kluczach = zastosowanie mapy.

Użyteczność map (2)

System "diagnostyczny":

```
 switch(symptom) {
   case "Symptom 1" : showMessageDialog(null, "Wykonaj dzialanie A"); break;
   case "Symptom 2" : showMessageDialog(null,"Wykonaj dzialanie ABC"); break;
   // ...
   case "Symptom 100" : showMessageDialog(null,"Wykonaj dzialanie XYZ"); break;
   default: showMessageDialog(null, "Nie znam tego symptomu");
}
```

Złe rozwiązanie: kod jest długi (w tym przykładzie ponad 100 linii) i nudny, poza tym wszelkie zmiany (np. dodanie nowych symptomów albo zmiany działań do wykonania) powodują konieczność modyfikacji kodu i ponownego kompilowania "systemu".

Właściwe rozwiązanie = zastosowanie mapy.

Użyteczność map (3)

System "diagnostyczny" - mapa tworzona z pliku TSV:

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

def dmap = new File('diagnose.tsv') // Przy okazji inny sposób
              .readLines()          // budowy mapy z danych w pliku
              .collectEntries { it.tokenize('\t') }

while (sympt = showInputDialog 'Enter symptom' ) {
  showMessageDialog null, dmap[sympt] ?: 'n/a'
}
```

Dane odseparowane od kodu,
kod zwięzły i prosty,
bez powtórzeń,
efektywny.

Użyteczność map (4)

Ale dlaczego dane są zapisane w takim formacie (tsv)? W przypadku "systemu diagnostycznego" lepiej użyć innych formatów lub bazy danych. O przetwarzaniu wygodnych standardowych formatów (XML, JSON, YAML) i o pracy z bazami danych oraz związanym z tym użyciu map będzie szczegółowo mowa w wykładzie 13.

Teraz – w kontekście użyteczności map - zobaczymy jak można zastosować format JSON. Dane "systemu diagnostycznego" w formacie JSON mogą być zapisane tak (plik diagnose.json):

```
{
   "Symptom 1": "Wykonaj dzialanie A",
   "Symptom 2": "Wykonaj dzialanie ABC",
   "Symptom 100": "Wykonaj dzialanie XYZ"
}
```

Wtedy kod "systemu diagnostycznego" jeszcze się uprości.

Użyteczność map (5)

Zastosujemy klasę JsonSlurper ze standardowej biblioteki Groovy (w JDK nie ma standardowych środków przetwarzania formatu JSON).
JsonSlurper służy do łatwego/bezpośredniego uzyskiwania struktur danych (takich jak mapy, listy, listy map itp.) z formatów JSON zapisanych w plikach lub też pozyskanych jako teksty (np. z bazy danych). Tu jedną linijką kodu, z pliku diagnose.json od razu uzyskamy mapę:

```
import static javax.swing.JOptionPane.*
import groovy.json.JsonSlurper

def dmap = new JsonSlurper().parse(new File('diagnose.json'))

while (sympt = showInputDialog 'Enter symptom' ) {
  showMessageDialog null, dmap[sympt] ?: 'n/a'
}
```

Użyteczność map (6)

Format JSON jest ważny, bo:

a) jest używany w web-serwisach
b) jest używany natywnie w JavaScript do reprezentacji obiektów
c) może też służyć do utrwalania obiektów w innych językach niż JS
d) jest używany w dokumentowych nierelacyjnych bazach danych

A z formatów JSON najczęsciej uzyskujemy mapy i w programie, łatwymi środkami języka Groovy, możemy elastycznie na nich działać.

Użyteczność map (7)

Przykład: bardzo prosty, choć ograniczony w użyciu, serwis fixer.io dostarcza kursów wybranych walut w formacie JSON (mapy).
Zob. składnię żądania GET serwisu.
Prosty kalkulator walutowy może wyglądac tak:

```
import groovy.json.JsonSlurper

def ask = { msg-> javax.swing.JOptionPane.showInputDialog(msg) }

def base = ask 'Enter base currency symbol'
def currs = ask('Currencies to calc').tokenize().join(',')
def xmap = new JsonSlurper().parse(
             new URL("https://api.fixer.io/latest?base=$base&symbols=$currs"))
def amt = ask('Enter money ammount').toInteger()
xmap.rates.each { k, v ->
  println "$amt $k = ${String.format('%.2f', amt/v)} $xmap.base"
}
```

Po wprowadzeniu jako base USD, a PLN i THB jako symboli walut oraz kwoty 500 możemy dostać (zależnie od aktualnego kursu):
500 PLN = 139,38 USD
500 THB = 15,14 USD

Użyteczność map (8)

Proste obiekty języka Groovy mogą być łatwo przekształcane w mapy i odwrotnie – mapy w obiekty. Pozwala to bardzo prosto zapisywać/odczytywać obiekty do/z plików (np. w formacie YAML czy JSON) oraz do/z dokumentowych baz danych.

Metoda getProperties() zwraca mapę, w której pod kluczami-nazwami właściwości znajdują się ich wartości:

```
package mapy

class Student {
  def indNr
  def name
}

// Uwaga: różne formy tworzenia obiektów – zob. wykład 11
def s = new Student(indNr: 's0001',  name: 'Abacki Adam')
def pmap = s.properties
println pmap
```

Wynik:

```
[indNr:s0001, class:class mapy.Student, name:Abacki Adam]
```

Użyteczność map (9)

Możemy więc zapisać zawartość obiektu do bazy danych lub do pliku w formacie JSON lub YAML, a później go odtworzyć. Na przykład:

Uwaga: klasa JsonBuilder z groovy.json pozwala na utrwalanie złożonych obiektów w formacie JSON

```
import org.yaml.snakeyaml.Yaml

class Student {
  def indNr
  def name
}
def s = new Student(indNr: 's0001',  name: 'Abacki Adam')
def pmap = s.properties
pmap.remove('class') // informacja o klasie nie jest nam potrzebna
// Zapis do pliku w formacie YAML z uzyciem biblioteki SnakeYAML
def yaml = new Yaml()
def yfile = new File('stud.yaml')
yfile.text = yaml.dump(pmap)
// Odtworzenie z pliku
def nmap = yaml.load(yfile.text)
println nmap
// Wygodna forma budowy obiektu przez przypisanie mapy (zob. w.11)
Student s2 = nmap mapy
println s2.name

Wynik:
[indNr:s0001, name:Abacki Adam]
Abacki Adam
```

Użyteczność map - podsumowanie

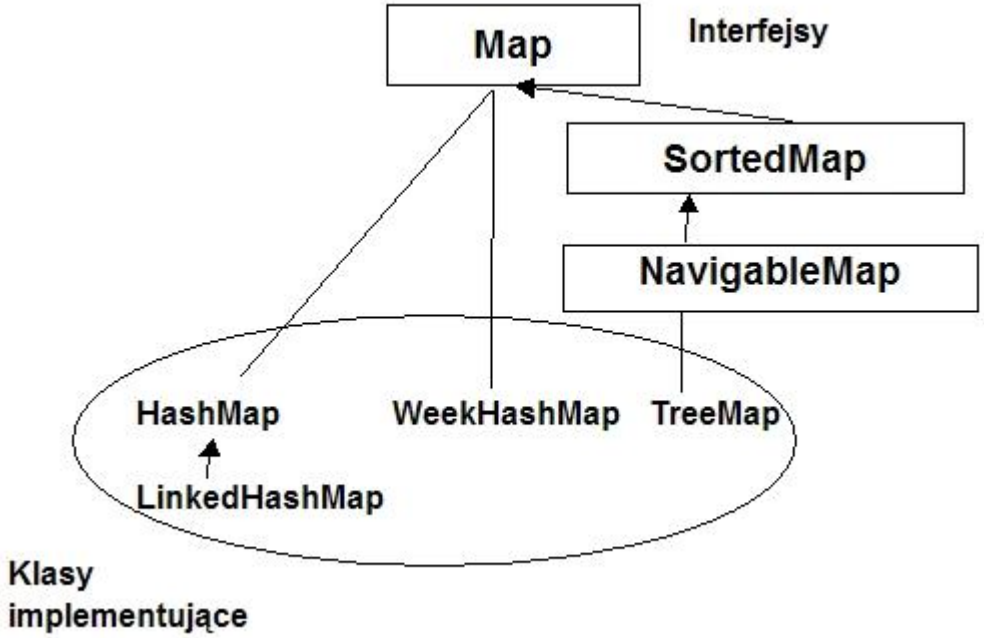
Istotą zastosowania map jest możliwość łatwego i jednocześnie szybkiego odnajdywania informacji w powiązanych zestawach danych, a także tworzenia zwięzłego, elastycznego i utrzymywalnego kodu oraz prostego przetwarzania informacji uzyskanych z serwisów web, dokumentowych baz danych, a ogólniej tworzenia i odczytywania treści w standardowych i wygodnych formatach JSON i YAML.

Dlatego warto poświęcić czas na zapoznanie się ze szczegółami działania na mapach.

W języku Groovy jest ono bardzo łatwe i przyjemne.

Rodzaje map

Hierarchia:



HashMap - klucze to HashSet

TreeMap - klucze to TreeMap

LinkedHashMap - klucze w kolejnosci w jakiej były dodawane

Tworzenie i inicjacja map

Pusta mapa:

```
map = [:] // map jest typu Map i LinkedHasMap
map = [:] as MapKlas // map jest typu MapKlas (np. HashMap lub TreeMap)
```

Inicjacja:

```
map = [ entry1, entry2, ..., entryN ]
```

gdzie *entry* to para:

key : value

bezpośredni napis;
jeśli stanowi liczbę, to będzie potraktowany jako liczba w przeciwnym razie jako String
albo
dowolne wyrażenie ujęte w nawiasy okrągłe (np. nazwa zmiennej) – wtedy klucz będzie miał wartośc tego wyrażenia

dowolne wyrażenie
- pod kluczem znajdzie się wartość tego wyrażenia

Inicjacja map: przykład 1

```
def show = { map->
  map.each { k, v ->
    println "$k = $v / key type is: " + k.getClass()
  }
}

map = [a : 1, b: 2]
show map

map = [1: 'a', 2: 'b']
show map

map = [1.1 : 'x' ]
show map
```

wynik:

```
a = 1 / key type is: class java.lang.String
b = 2 / key type is: class java.lang.String
1 = a / key type is: class java.lang.Integer
2 = b / key type is: class java.lang.Integer
1.1 = x / key type is: class java.math.BigDecimal
```

Inicjacja map - przykład 2

```
def func(n) {
  n*10
}

waw = 'Warszawa'
map = [Kraków: 'Małopolska', (waw) : 'Mazowsze',
       Mazowsze: waw, waw: 'Warszawa',
       'stolica Polski' : 'Warszawa',
       ('Miasto ' + waw) : 'Warszawa',
       (func(waw.size())) : waw.reverse() ]

map.each { println it } // it oznacza Map.Entry – parę klucz-wartość

Wynik:
Kraków=Małopolska
Warszawa=Mazowsze
Mazowsze=Warszawa
waw=Warszawa
stolica Polski=Warszawa
Miasto Warszawa=Warszawa
80=awazsraW
```

Inicjacja map – SPREAD MAP operator

Znany nam operator spread (**col;* por. wykład 8) w przypadku map umożliwia zastąpienie sekwencji dodawania do danej mapy wejść z innych map przez prostą inicjację. Operator ma tu formę **:***
i nazywa się SPREAD MAP

```
def psykoty = [ psy: 2, koty: 3 ]
def owcekozy = [ owce: 1, kozy: 1]

// mapa zwierzaków ?
def zwierzaki = [ psykoty, owcekozy] // nie! lista map
println zwierzaki

// użycie operatora SPREAD MAP (*:) daje mapę!
zwierzaki = [ *:psykoty, *:owcekozy ]
println zwierzaki
```

Wynik:

```
[[psy:2, koty:3], [owce:1, kozy:1]]
[psy:2, koty:3, owce:1, kozy:1]
```

Dodawanie do map operatory << oraz +

Operatory te działają tak jak w przypadku innych zestawów danych (z tym, że do mapy dodawane są wejścia z innej mapy lub bezpośrednio jedno takie wejście):

```
def map = [a: 1, b: 2]
def map1 = [c:3, d: 4]
println map + map1
println map
map << map1
println map
map += [x: 10, y: 20]
println map
def map2 = [kuba: 10,  barney: 9]
//map << map2.entrySet() // to nie działa
// ale to tak:
map2.entrySet().each { map << it }
println map
```

```
Wynik:
[a:1, b:2, c:3, d:4]
[a:1, b:2]
[a:1, b:2, c:3, d:4]
[a:1, b:2, c:3, d:4, x:10, y:20]
[a:1, b:2, c:3, d:4, x:10, y:20, kuba:10, barney:9]
```

Dostęp do wartości pod kluczami

Trzy możliwości: użycie kropki, indeksowania lub metody get()

```
def map = [a:1, 'chg commited':2, 1: 'a', 2: 'b']

def var = 'a'
// Użycie kropki - po kropce tylko klucz typu String
println map.a
println map.'chg commited'
println map.var // nie ma wartości pod kluczem 'var'
map.a = 100
map.'chg commited' = 3
println map

// Użycie indeksowania - indeks to dowolne wyrażenie dające wartość kluczq
def a = 1
println map[a]   // zmienna a (ma wartośc 1 i to będzie klucz)
println map['a'] // literał 'a' jest kluczem
println map[var] // kluczem będzie wartość zmiennej var
println map[1]   // liczba całkowita - klucz
map[a+1] = 100
println map
```

```
1
2
null
[a:100, chg commited:3, 1:a, 2:b]
a
100
100
a
[a:100, chg commited:3, 1:a, 2:100]
```

Metoda get

Java: get(key), od wersji 8: getOrDefault(defval)

Groovy: dodatkowo get(key, defval)

Uwaga: działanie metod z wartościami domyślnymi w Javie i Groovy jest różne (zob. poniżej)

```
map = [a: 1, b: 2 ]
println map.get('a')
println map.getOrDefault('w', 'val for key w') // Java 8
println map.get('z', 'val for key z') // Groovy: dodaje do mapy!
println map
println map.z // wartość z poprzedniego get już jest w mapie

// Wynik:
1
val for key w
val for key z
[a:1, b:2, z:val for key z]
val for key z
```

Metoda withDefault

Metoda withDefault(Closure) użyta wobec mapy zwraca nową mapę z ustalonymi wartościami domyślnymi (dla brakujących kluczy), określanymi przez kod domknięcia, w którym można wykorzystać przekazany jako argument klucz.

```
// Zliczenie liczby wystąpień słów w pliku
def map = [:].withDefault { 0 }
new File('tekst.txt').text.split( /[\s\p{Punct}]+/ ).each { map[it]++ }
map.each { println it }

// A tu domknięcie korzysta z wartości klucza
hash = [:].withDefault { key-> key.hashCode()}
println hash.a
println hash.Asia
println hash.Europe
```

Zawartość pliku tekst.txt:

ali kot nazywa się "pies",
ali pies nazywa się "kot"

```
Wynik:
ali=2
kot=2
nazywa=2
się=2
pies=2
97
2050282
2086969794
```

Kropka – uniwersalny akcesor

Warto zauważyć, że nieprzypadkowo wybrano kropkę jako sposób dostępu do wartości w mapie „po kluczach”. Jest to zgodne z konwencją obiektową dostępu do właściwości obiektów. Mapy blisko wiążą się z obiektami, możemy więc pisać fragmenty kodu, który w sposób uniwersalny działają i na mapach i na obiektach:

```groovy
import groovy.transform.*

@Canonical  // dodaje m.in. tuple-constructor i toString
class Dog {
  def name
  def ill   // czy jest chory?
}

def dog1 = new Dog('Barney', true)
def dog2 = [ name: 'Kuba', ill: true ]

cure(dog1, dog2)

println dog1
println dog2


def cure( ...dogs ) {
  dogs.each { it.ill = false }
}
```

Wynik:

```
Dog(Barney, false)
[name:Kuba, ill:false]
```

Kolekcje map, kropka i star-dot operator

W języku Groovy wprowadzono do składni elementy wyrażeń w stylu XPath (nazywa się to GPath). Zob. informacje o GPath w dokumentacji.

Dotyczy to kolekcji obiektów i map, elementów XML itp.

Gdy mamy kolekcje map c = [m1, m2, … mN], odwołanie c.key daje listę wartości spod klucza key ze wszystkich map. Podobnie działa znany nam już operator star-dot (spread-dot), z tą różnicą, że jeśli elementem kolekcji jest null, to w wynikowej liście na tym miejscu znajdzie się też null.

```
def mapList = [ [a: 1, b: 2],  [a: 10, b: 11],
                [a: 20, b: 21 ],  null ]
println mapList.a
println mapList*.a
println mapList.b
println mapList*.b

Wynik:
[1, 10, 20]
[1, 10, 20, null]
[2, 11, 21]
[2, 11, 21, null]
```

Przykład

W pliku TSV (z http://www.geonames.org/) znajdują się informacje o krajach i obszarach świata. Rodzaje informacji widać w poniższym kodzie w literale *keynames*. Daje on jednocześnie nazwy kluczy w mapie, opisującej informacje dla danego kraju. Całość (informacje o wszystkich krajach) zebrana jest na liście map.

```
def keynames = 'code name capital area popul continent'.split()

def list = new File('countries.tsv').readLines().collect {
  def map = [:]
  def data = it.split('\t')[0..keynames.size()-1]
  keynames.eachWithIndex { k, i ->  map[k] = data[i] }
  map
}
println list.size() // ile mamy panstw-obszarow

// Jakie mamy oznaczenia kontynentow?
def cont = list.continent as Set
println cont
// ile jest ludzi na świecie?
println list.popul*.toBigDecimal().sum()
```

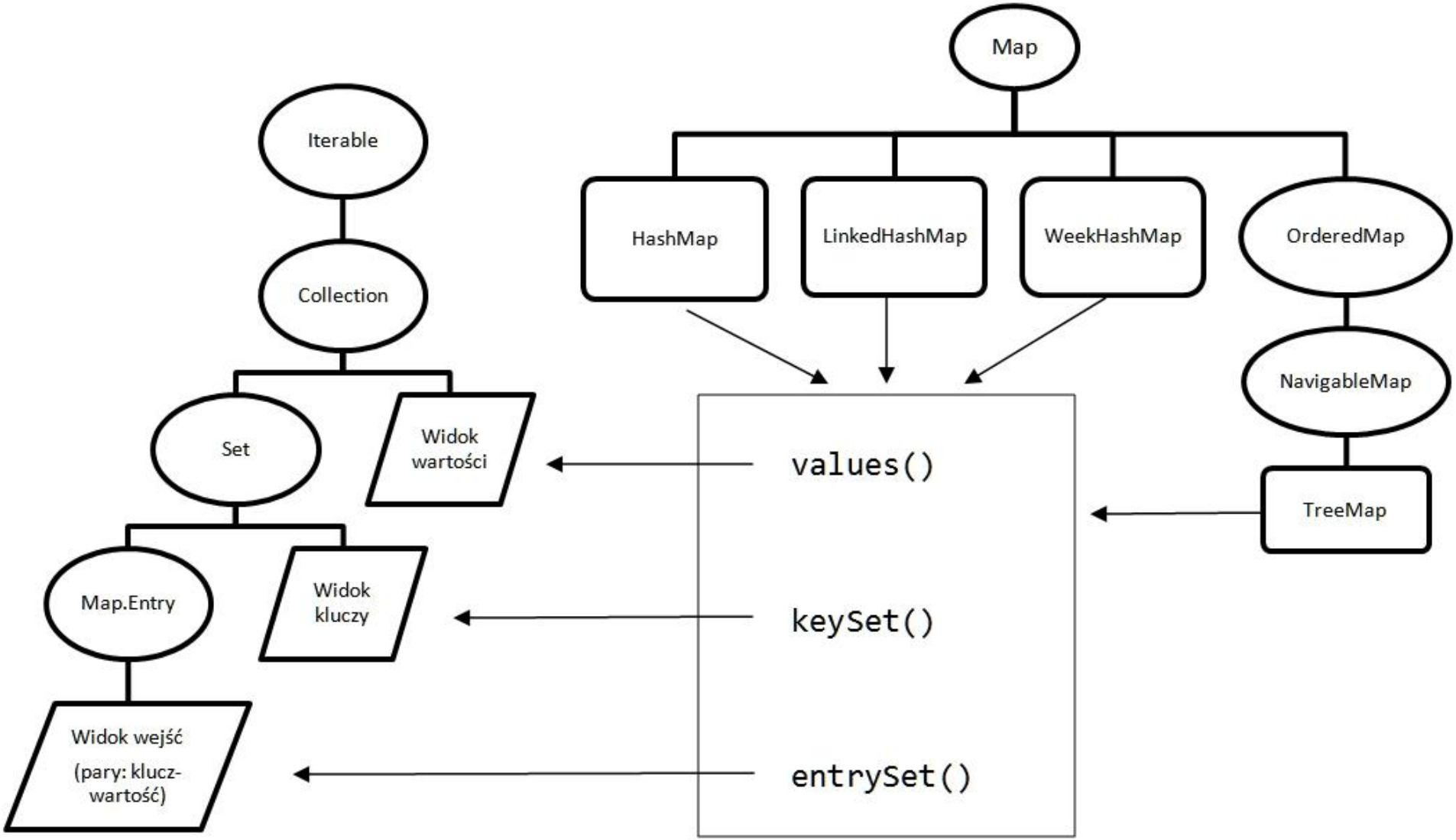
```
Wynik:
250
[EU, AS, NA, AF, AN, SA, OC]
6857911563
```

Standardowe metody interfejsu Map (JDK)

void **clear**()
Removes all of the mappings from this map (optional operation).
boolean **containsKey(Object key)**
Returns true if this map contains a mapping for the specified key.
boolean **containsValue(Object value)**
Returns true if this map maps one or more keys to the specified value.
Set<Map.Entry<K,V>> **entrySet()**
Returns a Set view of the mappings contained in this map.
V **get(Object key)**
Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
Boolean **isEmpty()**
Returns true if this map contains no key-value mappings.
Set<K> **keySet()**
Returns a Set view of the keys contained in this map.
V **put(K key, V value)**
Associates the specified value with the specified key in this map (optional operation).
void **putAll(Map<? extends K,? extends V> m)**
Copies all of the mappings from the specified map to this map (optional operation).
V **remove(Object key)**
Removes the mapping for a key from this map if it is present (optional operation).
Int **size()**
Returns the number of key-value mappings in this map.
Collection<V> **values()**
Returns a Collection view of the values contained in this map.

Groovy dodaje wiele nowych metod, m.in. te które są już zdefiniowane dla Iterable i Collection.

Mapy, widoki map a inne kolekcje



Iterowanie po mapach

```
map.each { entry ->  // entry.key == klucz, entry.value == value }
map.each { key, val -> ... }
map.eachWithIndex { entry, i -> ... } lub {k, v, i -> ...}
for (key in map.keySet()) ins
for (val in map.values()) ins
for (Type key : map.keySet() ins
for (Map.Entry e : map.entrySet()) ins
```

itd.

Wyszukiwanie w mapach

Dostępne metody:

findAll { entry -> ... } lub findAll { key, value -> ... }

Tak samo find, every , any ...

Przykład:

```
map = [ Polska : 38, Czechy: 7, Węgry: 7,
        Hiszpania: 45, Chiny: 2000, Indie: 1000 ]

res = map.findAll { it.value < 10 } // zwraca LinkedHashMap
println res.getClass()
println res

Wynik:
class java.util.LinkedHashMap
[Czechy:7, Węgry:7]
```

Mapy z grupowania

Metody groupBy… (z ogólnych klas Iterable, Collection etc = zob. dokumentacje JDK) pozwalają łatwo tworzyć mapy z kolekcji:

```
// lista wszystkich krajów
// elementy to listy zawierające info
// [ kod, nazwa,  ..., kontynent, ... ]
// kod kontynentu na tej liście ma indeks 5
def list = new File('countries.tsv').collect { it.split('\t') }
// grupujemy po kodach kontynentów
def map = list.groupBy { it[5] } // domknięcie daje klucz
// ile krajów/obszarów jest na każdym kontynencie
map.each { k, v ->
  println k + ' ' + v.size()
}
Wynik:
EU 53
AS 52
NA 41
AF 58
AN 5
SA 14
OC 27
```

Pamiętajmy również o metodach collectEntries, które też pozwalają tworzyć mapy z kolekcji.

Grupowanie map

Metody groupBy… stosowane wobec map pozwalają grupować wejścia w mapy lub listy wejść, umieszczając je pod kluczem zwracanym przez domknięcie:

```
map = [ Polska : 38, Czechy: 7, Węgry: 7,
        Hiszpania: 45, Chiny: 2000, Indie: 1000 ]

gmap = map.groupBy { k, v ->
  if (v < 10) 'small'
  else if (v < 100) 'medium'
  else 'big'
}
gmap.each { println it }
```

Wynik:

```
medium={Polska=38, Hiszpania=45}
small={Czechy=7, Węgry=7}
big={Chiny=2000, Indie=1000}
```

Sortowanie map

W przeciwieństwie do Javy, sortowanie map w Groovy wg dowolnych kryteriów (korzystających i z kluczy i wartości) jest bardzo łatwe. Metody sort zdefiniowane w interfejsie Map w GDK dają nam po temu środki (zobacz GDK).

Przykład.

```
// aby widziec jaka jest kolejnosc zapisu
infoFields = 'code name capital area popul continent'

// mapa [ nazwaKraju: ludność ]
map = new File('countries.tsv').readLines().collectEntries {
  data = it.split('\t')
  [data[1], data[4]]
}

res = map.sort { it.value.toInteger() }
println res.getClass()

es = res.entrySet() as List

// Duże:
es[-1..-5].each {   println it}

// Male:
es[1..5].each {   println it}
```

Wynik

```
class java.util.LinkedHashMap
China=1330044000
India=1173108018
United States=310232863
Indonesia=242968342
Brazil=201103330
Bouvet Island=0
Heard Island and McDonald Islands=0
United States Minor Outlying Islands=0
South Georgia and the South Sandwich Islands=30
Pitcairn=46
```