# Metody programowania

# Wprowadzenie do platformy Java

Dr inż. Andrzej Grosser

*Częstochowa, 2013*

# Spis treści

**1. Kolekcje** **5**

1.1. Tablice . . . . . 5

1.1.1. Tablice wielowymiarowe . . . . . 7

1.2. Łańcuchy znaków . . . . . 8

1.3. Typy wyliczeniowe . . . . . 10

1.4. Inne kolekcje . . . . . 12

1.4.1. Interfejsy kolekcji . . . . . 12

1.4.2. Implementacje interfejsów kolekcji . . . . . 13

1.5. Iterator . . . . . 13

**Literatura** **17**

# 1. Kolekcje

Do tej pory nie pokazano jeszcze wszystkich sposobów tworzenia zmiennych grupujących wiele wartości. Niniejszy rozdział ma wypełnić tą lukę. Skoncentrowano się w nim na tablicach, łańcuchach znaków, typach wyliczeniowych. Wspomniano także o pozostałych kolekcjach z biblioteki standardowej, ale bez wchodzenia w szczegóły techniczne (można je poznać z bogatej dokumentacji biblioteki standardowej Javy). Do napisania tego rozdziału użyto książki "Core Java"[1] i dokumentacji Javy.

## 1.1. Tablice

Tablice są strukturą danych przechowującą wartości tego samego typu. Dostęp do poszczególnych elementów jest realizowany zazwyczaj za pomocą operatora indeksowania. Elementy są z reguły numerowane od 0 (są języki dopuszczające numerowanie indeksów tablicy od innych wartości np. Pascal).

Tablice deklaruje się w Javie używając typu przechowywanych wartości, po którym następują dwa nawiasy kwadratowe i nazwa tablicy (można też nawiasy kwadratowe zapisać po nazwie tablicy). Na przykład:

```
1 double[] tab;
2 //lub analogicznie
3 double tab[];
```

Tak jak napisano wcześniej, pokazany powyżej kod jest jedynie deklaracją tablicy, należy jeszcze przydzielić pamięć dla elementów, używa się do tego celu operatora `new` podając za nim typ elementów i w nawiasach kwadratowych rozmiar tworzonej tablicy:

```
1 double[] tab1 = new double[10];
2 int rozmiar = 100;
3 int[] tab2 = new int[rozmiar];
```

Na listingu powyżej zostanie utworzona tablica tab1 10 elementów typu rzeczywistego podwójnej precyzji i tablica tab2 liczb całkowitych o rozmiarze 100. Jak można zauważyć rozmiar nie musi być stałą (czyli tablice Javy są prawie dokładnym odpowiednikiem tablic

dynamicznych z C++ - zawsze tworzone są na stercie, w prawie ten sam sposób; nie trzeba jedynie pamiętać o zwalnianiu pamięci - to w Javie spada na automatyczny odśmiecacz pamięci).

Dostęp do elementów jest realizowany za pomocą operatora indeksowania dokładnie tak samo jak w C++, elementy numerowane są od 0 do rozmiar - 1. Przy czym w przeciwieństwie do C++ tablice Javy znają swój rozmiar, można go poznać w trakcie uruchomienia programu - używając konstrukcji - `nazwa_tablicy.length`. Ponadto odwołanie się do elementu o indeksie spoza dozwolonego zakresu powoduje powstawanie wyjątku - `java.lang.ArrayIndexOutOfBoundsException`. Przykładowo dostęp do elementów tablicy tab jest realizowany następująco:

```
1 double tab[] = new double[20]
2 for(int i = 0; i < tab.lentgth; ++i) {
3     System.out.println(tab[i]);
4 }
5 //lub analogicznie korzystając dodatkowej
6 //składni for
7 for(int elem : tab) {
8     System.out.println(elem);
9 }
```

Elementom tablicy można nadać wartości początkowe korzystając z tzw. inicjalizatorów tablic - podobnie jak w C++ podaje się w nawiasach klamrowych wartości kolejnych elementów. Na przykład:

```
1  int[] tab[] = {1,2,3,4,5};
2  // lub
3  int tab[] = new int[]{1,2,3,4,5};
```

Jak można zauważyć w przypadku pierwszej konstrukcji nie ma potrzeby korzystania z operatora new. Ponadto, jak widać, nie były zapisywane rozmiary tablic - kompilator na podstawie liczby zapisanych elementów sam wyznacza długość tablicy.

Tworzenie niezależnej kopii tablicy wymaga użycia metod zewnętrznych. Wynika to z tego, że nazwa tablicy jest jedynie odniesieniem do właściwej struktury danych, przypisanie, więc spowoduje, że dwie zmienne będą pokazywały ten sam obszar pamięci (dokładnie taki sam problem jak dla rozważanych wcześniej obiektów klas). Na przykład:

```
1 int[] tab1 = {1,2,3,4,5};
2 int[] tab2 = tab1;
```

```
3 tab2[0] = 10;
```

W podanym powyżej przypadku tab2 pokazuje na tą samą tablicę co tab1, dlatego przypisanie z ostatniej linijki kodu spowoduje, że tab1[0] zwróci 10 a nie 1, jak można byłoby się spodziewać na pierwszy rzut oka. Dlatego jeżeli należy przechowywać dwie osobne tablice, należy użyć metody `copyOf` z klasy pomocniczej `Arrays`:

```
1 int[] tab1 = {1,2,3,4,5};
2 int[] tab2 = Arrays.copyOf(tab1, tab1.lentgth);
3 tab2[0] = 10;
```

Dla pokazanego powyżej kodu przypisanie z ostatniej linijki spowoduje jedynie modyfikacje elementu tablicy pokazywanej przez tab2.

### 1.1.1. Tablice wielowymiarowe

Java, jak każdy praktycznie stosowany język programowania ogólnego przeznaczenia, oferuje również tworzenie tablic wielowymiarowych. Można to zrobić korzystając z inicjalizatorów tablic wielowymiarowych:

```
1 int[][] mac = {
2     {1,2,3},
3     {4,5,6},
4     {7,8,9}
5 };
```

W powyższym kodzie kolejne wiersze z macierzy są inicjowane wartościami zapisanymi pomiędzy wewnętrznymi nawiasami kwadratowymi.

Tablice wielowymiarowe można też tworzyć za pomocą operatora `new`. Tablicę tworzy się wtedy podając kolejne wymiary w nawiasach kwadratowych. Dla przykładu dla tablicy o wymiarze 10 na 10 należy użyć konstrukcji (każdy wiersz będzie miał dokładnie 10 elementów):

```
1 int[][] tab = new int[10][10];
```

Możliwa jest też konstrukcja kilku etapowa taka jak w C++, na przykład dla tablicy dwuwymiarowej, tworzy się najpierw tablicę przechowującą poszczególne wiersze a dopiero później przydziela się dla nich pamięć (wiersze nie muszą mieć tyle samo elementów):

```
1 double tab[][] = new double [10][];
2 for(int i = 0; i < tab.length; ++i)
3     tab[i] = new double[i + 1];
```

W powyższym kodzie zostanie utworzona macierz trójkątna.

Dostęp do elementów tablicy wielowymiarowej jest realizowany w Javie tak samo jak w C++ - po nazwie tablicy w nawiasach kwadratowych podaje się numery poszczególnych indeksów, na przykład:

```
int tab[][] = {
  {1,2,3},
  {4,5,6},
  {7,8,9}
};
for(int i = 0; i < tab.lentgth; ++i)
   for(int j = 0; j < tab[i].length; ++j) {
      System.out.println(tab[i][j]);
   }
```

Gdy konieczny jest dostęp do każdego elementu tablicy po kolei można również wykorzystać konstrukcję:

```
for(int[] wiersz : tab)
   for(int element : wiersz)
      System.out.println(elem);
```

## 1.2. Łańcuchy znaków

Do przechowywania łańcuchów znaków służą w Javie obiekty typu `String`. Przechowywane są w nich znaki Unicode (dwu bajtowe). Tworzy się je najczęściej bezpośrednio, bez użycia operatora new:

```
String s1;
//Odpowiednik
//String s1 = new String();
```

Zmienna s1 ma wartość łańcucha pustego - nie należy jej mylić z obiektem nie wskazującym na wartość (null). Na przykład:

```
String s1;
String s2 = null;
int r1 = s1.length();
int r2 = s2.length();
```

Dla powyższego kodu wywołanie metody length() (zwracającej rozmiar łańcucha) zakończy się zwróceniem wartości 0 dla obiektu s1, zaś wyjątkiem dla obiektu s2.

Obiekty typu String mogą być inicjowane wartościami literału łańcuchowego zapisanego pomiędzy dwoma cudzysłowami:

```
String s2 = "";
String s3 = "Ala ma kota";
```

Konkatenacja łańcuchów znaków jest realizowana z wykorzystaniem operatora +, na przykład:

```
String s1 = "Witaj ";
String s2 = "świecie"
System.out.println(s1 + s2 + "!";
```

Napis wynikowy jest złożeniem wartości kolejnych operandów (dlatego w powyższym przykładzie uzyska się na ekranie Witaj świecie!. Oprócz łączenia łańcuchów znaków możliwa jest także konkatenacja łańcucha ze zmienną, która nie jest łańcuchem znaków (jest ona konwertowana do łańcucha przed połączeniem):

```
int miejsce = 1;
String str = "miejsce"
String wyn = 1 + str;
//Napis wyn na wartość: 1miejsce
```

Łańcuchy znaków typu String są niezmienne, nie można zmodyfikować wartości znaku. Poniższy kod jest błędny w Javie:

```
String str = "xxx";
str[0] = "y";
```

W powyższym przypadku jest możliwa jedynie zmiana całości łańcucha znaków (tworzony jest nowy obiekt)[1]:

```
String str = "xxx"
str = "y" + str.substring(1);
```

Nie można użyć do porównywania wartości użyć operator równości, gdyż porównuje on jedynie wartości odniesień, czyli innymi słowy adresy obiektów, a nie wartości kolejnych znaków z łańcuchów, na przykład:

```
String s1 = "kwadrat";
String s2 = "kwadrat";
if(s1 == s2) {
    System.out.println("Równe napisy");
}
```

[1]Metoda substring zwraca podłańcuch zaczynając od indeksu podanego argumentem.

W powyższym kodzie porównanie wartości za pomocą operator da wartość fałszu, gdyż są to dwa różne obiekty, w różnych obszarach pamięci (dla operatora równości nie ma znaczenia, że przechowują ten sam łańcuch znaków).

Porównywanie łańcuchów znaków jest realizowane przez metodę z klasy String - `equals` w ten sposób porównuje się ze sobą leksykograficznie kolejne elementy z dwóch łańcuchów znaków:

```
String imie1 = "Wojciech";
String imie2 = "Wojciech"
String imie3 = "Mariusz";
bool b1 = imie1.equals(imie2);
bool b2 = imie1.equals(imie3);
```

W powyższym kodzie przy pierwszym wywołaniu metoda `equals` zwróci wartość prawdy, natomiast przy drugim fałsz (łańcuchy są po prostu różne).

Oprócz wymienionych powyżej metod klasa String posiada też inne (np. trim() - metoda usuwająca białe znaki z początku i końca łańcucha). Opis ich znajduje się w dokumentacji, więc nie będą tutaj dokładnie omawiane.

## 1.3. Typy wyliczeniowe

Typy wyliczeniowe są specjalnym typem, który umożliwia przechowywanie w zmiennej predefiniowanego zbioru wartości. Są użyteczne w sytuacji, gdy potrzebny jest dla zmiennej ściśle określony zestaw stałych.

W Javie definicję typu wyliczeniowego poprzedza się słowem kluczowym `enum`, po którym następuje nazwa tego typu i w nawiasach klamrowych wyliczenie kolejnych wartości rozdzielonych przecinkami, ostatnią wartość kończy się średnikiem:

```
public enum DzienTygodnia {
  PONIEDZIAŁEK, WTOREK, ŚRODA,
  CZWARTEK, PIĄTEK, SOBOTA,
  NIEDZIELA;
}
```

Nieprzypadkowo przedstawiona składnia przypomina definicję klasy - w Javie definicje typów wyliczeniowych mogą zawierać metody i pola składowe:

```
public enum PlanetaWewnetrzna {
    MERKURY (3.303e+23, 2.4397e6),
```

```
3       WENUS    (4.869e+24, 6.0518e6),
4       ZIEMIA    (5.976e+24, 6.37814e6),
5       MARS     (6.421e+23, 3.3972e6);
6
7       private final double masa;
8       private final double promien;
9       Planet(double m, double r) {
10          masa = m;
11          promien = r;
12      }
13      private double masa() { return masa; }
14      private double promien() { return promien; }
15  }
```

Są więc traktowane prawie jak klasa - przy czym kompilator dodaje kilka metod do typów wyliczeniowych (np. `values()` - zwraca tablicę stałych zdefiniowanych w typie wyliczeniowym w porządku deklaracji). Ponadto nie można dziedziczyć własności po jakiejś klasie. Wynika to z tego, że typy wyliczeniowe dziedziczą niejawnie po java.lang.Enum a nie można przecież dziedziczyć po więcej niż jednej klasie bazowej. Java nie pozwala też na definiowanie metod i pól składowych przed wyliczeniem stałych.

Wartości typu wyliczeniowego tworzy się przypisując wybraną stałą do zmiennej, na przykład:

```
1 DzienTygodnia dzien = DzienTygodnia.PONIEDZIAŁEK;
2 PlanetaWewnetrzna planeta;
3 planeta = PlanetaWewnetrzna.ZIEMIA;
```

Do testowania wartości typu wyliczeniowego używa się najczęściej instrukcji wyboru:

```
1  class Test {
2    DzienTygodnia dzien;
3    public void zwroc() {
4       switch(dzien) {
5          case SOBOTA:
6          case NIEDZIELA:
7                System.out.println("Weekend");
8                break;
9          default:
10               System.out.println("Dzień roboczy");
11               break;
12      }
13   }
14 }
```

## 1.4. Inne kolekcje

W codziennej pracy z Javą może być konieczne użycie bardziej wyrafinowanych struktur danych. Pomocne mogą być tutaj generyczne kolekcje zdefiniowane w Javie. Użycie ich jest podobne jak szablonów z biblioteki standardowej C++ - po nazwie typu generycznego podaje się w nawiasach trójkątnych typ elementów:

```
1 ArrayList<Double> tab = new ArrayList<Double>();
2 //Można i tak:
3 ArrayList<Double> tab = new ArrayList<>();
```

W powyższym kodzie zostanie utworzony obiekt klasy generycznej `ArrayList` przechowujący obiekty klasy `Double` - opakowania na liczby rzeczywiste podwójnej precyzji. Nazwa typu została tutaj zapisana poprawnie - typy generyczne mogą być rozwijane jedynie dla obiektów wskazywanych przez referencje - typy podstawowe nie spełniają tego warunku, więc muszą być przechowywane z wykorzystaniem obiektowego opakowania. Nazwy takich typów wyróżniają się (z jednym małym wyjątkiem) wielką literą od zwykłych typów podstawowych (np. Double - double, Long - long, Boolean - boolean, Integer - int itp.).

### 1.4.1. Interfejsy kolekcji

Biblioteka standardowa Javy odseparowuje interfejs kolekcji od ich implementacji. Interfejsy dostarczają zestaw operacji jaki będzie konieczny do implementacji przez konkretne klasy, żeby mogły być one traktowane jako dana kolekcja.

Spośród kilku interfejsów kolekcji należy wyróżnić:

1. `Collection` - jest bazowym interfejsem dla kolekcji Javy, po którym dziedziczą pozostałe, wymusza implementację kilku metod, z których najważniejszymi są `add()` - dodawanie elementu do kolekcji i `iterator()` - zwracające obiekt implementujący interfejs Iterator (pośrednik do elementu w kolekcji, ukrywający szczegóły implementacji tego dostępu).
2. `List` - są sekwencją elementów, użytkownik może dokładnie określić, gdzie może wstawiać elementy. Dostęp do elementów jest realizowany przez jego indeks (liczba całkowita).

3. `Queue` - implementują ją obiekty, w których elementy są przechowywane do dalszego przetwarzania - muszą dostarczać operacji wstawiania, wyciągania i przeglądania elementów - `offer()`, `peek()` i `poll()`, `element()`,
4. `Deque` - szczególny przypadek kolejki, obiekty klas implementujących ten interfejs są liniowymi kolekcjami, do których można wstawić nowe elementy na ich początku i końcu.
5. `Map` - klasy implementujące ten interfejs umożliwiają tworzenie obiektów, w których klucze są stowarzyszone z przynajmniej jedną wartością. Map dostarcza trzy perspektywy z których można spojrzeć na kolekcję: jako zbiór kluczy, zbiór wartości i relacja klucz - wartość.
6. `Set` -są kolekcją elementów, która nie przechowuje duplikatów elementów.
7. `Stack` - oferuje możliwość wstawiania i ściągania obiektów ze stosu (wymaga implementacji metod `push()` i `pop()`).

### 1.4.2. Implementacje interfejsów kolekcji

Tabela 1.1 zawiera najważniejsze kolekcje z biblioteki standardowej Javy z krótkim opisem. Oczywiście, jak zwykle dokładny opis konkretnej klasy można znaleźć w dokumentacji. Czasem wystarczy wiedzieć o istnieniu danej rzeczy, żeby później można było z niej skorzystać.

## 1.5. Iterator

Do dostępu do elementów kolekcji, oprócz metod zdefiniowanych w ich klasach, używa się także iteratorów. Jak wspomniano wcześniej w Javie jest dostarczony dla iteratorów specjalny interfejs `Iterator`. Każda klasa, która go implementuje musi dostarczyć trzy metody:

1. `next()` - zwraca następny element w iteracji, gdy takiego nie ma, jest wyrzucany wyjątek.
2. `hasNext()` - zwraca prawdę, jeżeli w kolejnej iteracji można pobrać następny element, użycie jej przed next() zapobiega powstawaniu wyjątku.
3. `remove()` - usuwa element zwrócony przez swój iterator.

**Tabela 1.1.** Najważniejsze kolekcje generyczne Javy

| Rodzaj kolekcji | Krótki opis |
|---|---|
| `ArrayDeque` | Rozszerzalna implementacja interfejsu `Deque` |
| `ArrayList` | Rozszerzalna implementacja interfejsu `List` |
| `EnumMap` | Implementacja interfejsu `Map` dla kluczy typów wyliczeniowych |
| `EnumSet` | Implementacja interfejsu `Set` dla kluczy typów wyliczeniowych |
| `HashMap` | Oparta na tablicach implementacja interfejsu `Map` |
| `HashSet` | Oparta na `HashMap` implementacja interfejsu `Set` |
| `IdentityHashMap` | Implementacja interfejsu `Map`, w której wartości są porównywane za pomocą operator równości (`==`) |
| `LinkedHashMap` | Implementacja interfejsu `Map`, w której można przewidzieć porządek iterowania. |
| `LinkedHashSet` | Implementacja interfejsu `Set`, w której można przewidzieć porządek iterowania. |
| `LinkedList` | Oparta na listach dwukierunkowych implementacja interfejsu `List` |
| `PriorityQueue` | Oparta na stecie priorytetowej implementacja interfejsu `Queue` |
| `TreeMap` | Oparta na drzewach czerwono-czarnych implementacja interfejsu `Map` |
| `TreeSet` | Oparta na `TreeMap` implementacja interfejsu `NavigableSet` |
| `WeakHashMap` | Implementacja interfejsu `Map` z słabymi kluczami (klucze są usuwane, gdy nie są używane). |

Przykładowa interakcja z dostępem do elementów przez iteratory wygląda następująco:

```
1  ArrayList<Integer> tab = new ArrayList<>();
2  //Tutaj elementy są dodawane do tab
3  for(int i = 0; i < 10; ++i)
4      tab.add(i);
5  //Używanie interatorów
6  Iterator<Integer> iter = tab.iterator();
7  while(iter.hasNext())
8  {
9      Integer elem = iter.next();
10     //Operacja na elemencie
11     System.out.println(elem);
12 }
```

Jak widać jest to dość rozbudowany kod, dlatego wygodniej jest użyć konstrukcji wprowadzonej w Javie 1.5 (ukrywane są szczegóły implementacji):

```
1 for(Integer elem : tab) {
2   //Operacja na elemencie
3   System.out.println(elem);
4 }
```

Kompilator niejawnie przekształci taką pętlę do pętli korzystającej z iteratorów. Tego typu pętla jest możliwa dla obiektów implementujących interfejs `Iterable` (klasa musi, implementując ten interfejs, dostarczyć metodę `iterator()`).

# Literatura

[1] C. S. Horstmann and G. Cornell. *Core Java Volume I–Fundamentals*. Prentice Hall, 2011.