

Stefan Sokołowski

PROGRAMOWANIE OBIEKTOWE II

wykład 6

Inst. Informatyki Stosowanej, ANS
Elbląg, 2024/2025

Wykład 6, str. 1

Filtrowanie list

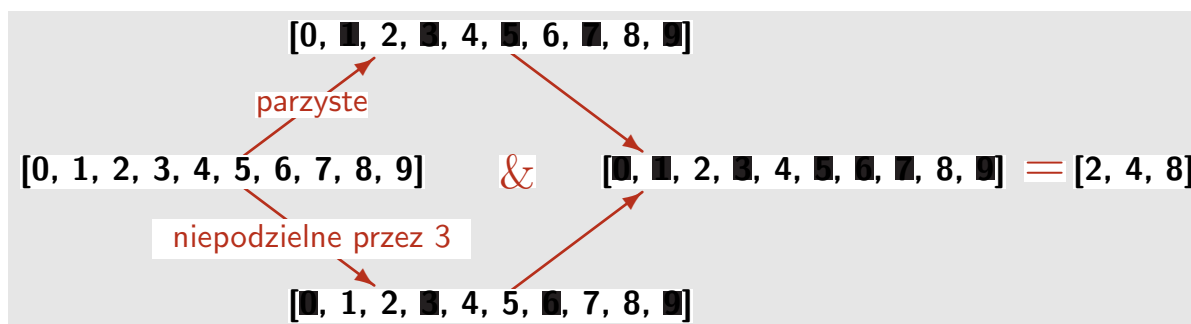
OGÓLNIEJSZE

→ Porównanie — od par do wartości logicznych

→ Własność — od pojedynczych elementów do wartości logicznych



Własnościami można na przykład filtrować listy:



```
podzPrzez2.koniunkcja(podzPrzez3.negacja())
```

Filtrowanie list

```
interface Wlasnosc<Typ> {
    public Boolean ff(Typ el);

    public default Wlasnosc<Typ> negacja() {
        return el -> !this.ff(el);
    }

    public default Wlasnosc<Typ> koniunkcja(Wlasnosc wl) {
        return el -> this.ff(el) && wl.ff(el);
    }
}
```

default

default

W nowszych wersjach Jawy (począwszy od Java 8) interfejs może zawierać metody **domyślne**. Klasy implementujące nie muszą (choć mogą) je definiować (przedefiniowywać).

Filtrowanie list

```
class Lista<Typ> extends ArrayList<Typ> {
    public Lista() { super(); }
    public Lista(Typ[] a) { super(Arrays.asList(a)); }

    public Lista filter(Wlasnosc wl) {
        Lista ls = new Lista();
        for (Typ el : this)
            if (wl.ff(el)) ls.add(el);
        return ls;
    }
}
```

konstruktory

dołączanie elementów spełniających wl.ff

Filtrowanie list

Z własnościami nie musimy sami się męczyć, bo...

Java zawiera interfejs `Predicate<Typ>` analogiczny do powyższego `Wlasnosc<Typ>`.

Klasa `ArrayList<Typ>` zawiera filtrowanie listy przez predykat: metoda `removeIf` działa dualnie do powyższej `filter` — włącza do listy wynikowej elementy **niespełniające** podanego predykatu (czyli **spełniające** usuwa).

iis.ans-elblag.pl/~stefan/Dydaktyka/Prog0biII/Prog06/P2_Filtry.java

Proszę przeglądać dokumentację!

W każdej nowej klasie...

```
class Punkt {
    private double xx, yy;
    public Punkt(double x, double y) {
        xx = x; yy = y;
    }
}

class Test {
    public static void main(String[] args) {
        Punkt pt1 = new Punkt(2,3);
        Punkt pt2 = new Punkt(2,3);
        System.out.println(pt1 == pt2);
        System.out.println(pt1.equals(pt2));
    }
}
```

Program tworzy dwa **identyczne** punkty i sprawdza ich równość przez `==` i przez `equals()`.

Wydruk: `false`
`false`

Wniosek:
Dla Jawy dwa osobno stworzone obiekty **nie są równe**.

We własnej klasie potrzebna jest własna metoda `equals()`.

W każdej nowej klasie...

Każde dwa obiekty można porównać operatorem `==` i metodą `equals()`. Ale porównywane są **miejsca w pamięci**, gdzie te obiekty „mieszkają”, a nie ich struktura wewnętrzna. Najczęściej warto jest metodę `equals()` **przykryć** własną metodą specyficzną dla danej klasy.

Teraz wydruk:

```
false
true
```

Wniosek:

Operator `==` nadal porównuje „miejsca zamieszkania”, ale metoda `equals()` porównuje wartości.

```
class Punkt {
    private double xx, yy;
    public Punkt(double x, double y) {
        xx = x; yy = y;
    }
    public double x() { return xx; }
    public double y() { return yy; }
    public Boolean equals(Punkt pt) {
        return pt.x()==xx && pt.y()==yy;
    }
}

class Test {
    public static void main(String[] args) {
        Punkt pt1 = new Punkt(2,3);
        Punkt pt2 = new Punkt(2,3);
        System.out.println(pt1 == pt2);
        System.out.println(pt1.equals(pt2));
    }
}
```

W każdej nowej klasie...

```
class Punkt {
    private double xx, yy;
    public Punkt(double x, double y) {
        xx = x; yy = y;
    }
}

class Test {
    public static void main(String[] args) {
        Punkt pt = new Punkt(2,3);
        System.out.println(pt);
    }
}
```

Program tworzy punkt i go drukuje. Wydruk:

```
Punkt@512ddf17
```

Wniosek:

W Java format wydruku obiektu z własnoręcznie zdefiniowanej klasy jest **nieczytelny**.

We własnej klasie potrzebna jest własna metoda `toString()`.

W każdej nowej klasie...

Na każdym obiekcie określona jest metoda `toString()` używana przez drukowanie systemowe; tak więc każdy obiekt da się wydrukować. Ale ten wydruk jest nieczytelny i prawie zawsze bezużyteczny. Najczęściej warto jest metodę `toString()` **przykryć** własną metodą specyficzną dla danej klasy.

Teraz wydruk:

(2.0, 3.0)

```
class Punkt {
    private double xx, yy;
    public Punkt(double x, double y) {
        xx = x; yy = y;
    }
    public String toString() {
        return "(" + xx + ", " + yy + ")";
    }
}

class Test {
    public static void main(String[] args) {
        Punkt pt = new Punkt(2,3);
        System.out.println(pt);
    }
}
```

W każdej nowej klasie...

W każdej nowej klasie $\mathcal{K}$ warto

- zdefiniować **własną** metodę

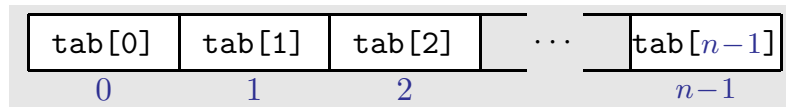
`Boolean equals( $\mathcal{K}$  obiekt)`

- zdefiniować **własną** metodę

`String toString()`

Tablice

```
Typ[] tab = new Typ[n];
```



$n = \text{tab.length}$ komórek o indeksach od 0 do $n - 1$
do każdej można wpisać obiekt klasy (typu) Typ

Na przykład:

- ① `String[] c = new String[5];` — tablica 5 napisów
- ① `c = new String[]{"C", "Java"};` — tablica 2 napisów
- ② `c[2] = "Python";` — **BŁĄD**

Po ① tablica c już **nie ma** 5 elementów, tylko 2.

Tablica tablic

```
Typ[][] tab = new Typ[n][];
```

<code>int[][] tab = new int[4][];</code>		tab:
<code>tab[0] = new int[]{ 0, 1, 2};</code>	→	0 0 1 2
<code>tab[1] = new int[]{10, 11};</code>	→	1 10 11
<code>tab[2] = new int[]{20, 21, 22, 23};</code>	→	2 20 21 22 23
<code>tab[3] = new int[]{};</code>	→	3

Tablica 2-wymiarowa = tablica tablic 1-wymiarowych

Jej wiersze **nie muszą** mieć tej samej długości!

iis.ans-elblog.pl/~stefan/Dydaktyka/ProgObiII/Prog06/P3_TablicaTablic.java

Tablice

- szybki dostęp do elementu zadanego indeksem;
- brak możliwości powiększenia lub zmniejszenia;
- trudne wstawianie w środek i usuwanie ze środka (trzeba przesuwać, nie zawsze się da):



Kolekcje

Jak w jedno miejsce pozbierać wiele danych tego samego typu?

Tablice nie są najlepszym sposobem realizacji pojęcia **zbioru**...

Operacje na **zbiorach elementów** które mogą być przydatne:

	tablice	interface Collection
dodać element	nie ma można wymienić na inny	zbiór. add (element)
usunąć element	nie ma można zaznaczyć nieistotność	zbiór. remove (element)
sprawdzić należenie	trzeba przeszukać	zbiór. contains (element)
moc zbioru (liczba elementów)	zwykle trzeba przeszukać omijając nieistotne	zbiór. size ()
coś zrobić na wszystkich elementach	for (Typ elem : tablica) { ... }	<i>iterator</i> (dalej wyjaśnione)

iis.ans-elblag.pl/~stefan/Dydaktyka/Prog0biIII/Prog06/P4.PrzeglądTablicy.java

Iterator

```
interface Iterator<T> {
    // czy istnieje jeszcze cos nieprzejrzanego:
    Boolean hasNext();
    // jesli istnieje, to go dawaj; inaczej sygnal bledu:
    T next();
}
```

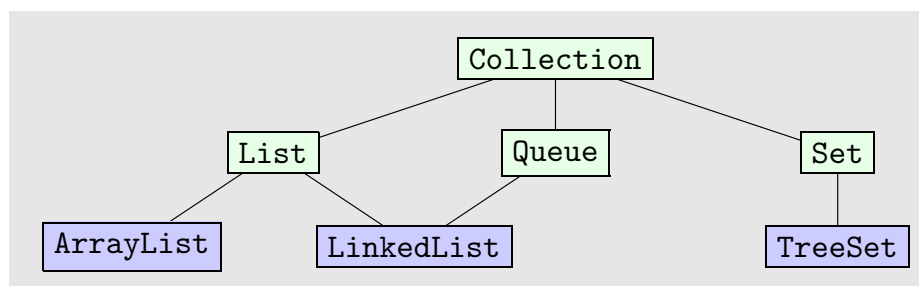
użycie iteratora	
jawne	niejawne
<pre>Iterator<T> it = zbiór.iterator(); while (it.hasNext()) akcja na elemencie it.next();</pre>	<pre>for (T elem : zbiór) akcja na elemencie elem;</pre>

To działa dla **wszystkich** klas implementujących **Collection**.

iis.ans-elblog.pl/~stefan/Dydaktyka/Prog0biII/Prog06/P5-Iteracja.java

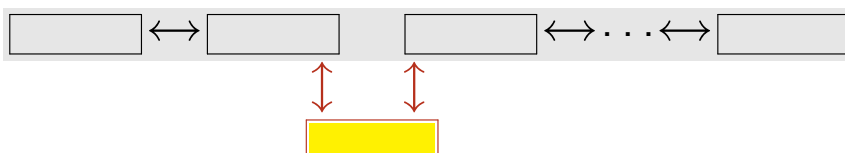
Czyli dla jakich klas?

Klasy implementujące Collection (przykłady)



- ArrayList** — jak tablice, ale automat. zmiana rozmiaru, wstawianie i usuwanie;
Uwaga: `ls[i]` nie działa, trzeba stosować `ls.get(i)` oraz `ls.set(i,x)` ;

- LinkedList** — 
 - powolny dostęp do elementu zadanego indeksem (przeszukiwanie listy);
 - powiększanie lub zmniejszanie listy bez ograniczeń;
 - łatwe wstawianie w środek i usuwanie ze środka:



Zwykłe tablice, czy listy wiązane, czy listy tablicowe?

Jakie zbiory do jakiego problemu programistycznego?

	zwykłe tablice	listy wiązane	listy tablicowe
stała liczba i ustalona kolejność elementów	✓		
pytania o element o dowolnym indeksie	✓		✓
zmienna liczba elementów		✓	✓
wstawienia i usunięcia tylko z początku lub końca		✓	✓
wstawienia i usunięcia w środku		✓	