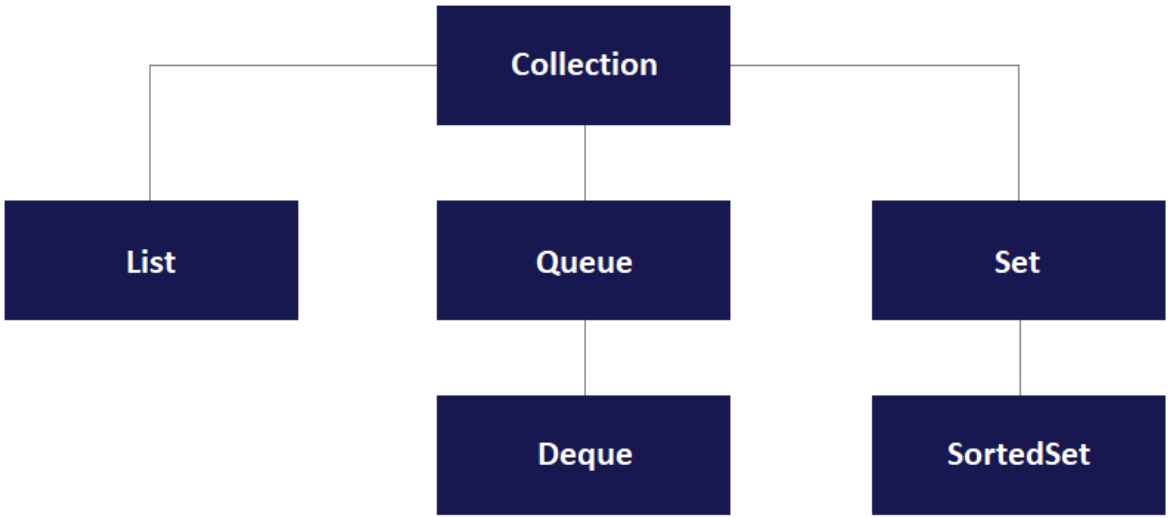
PROGRAMOWANIE OBIEKTOWE JAVA – LABORATORIUM

LISTY I SETY

Kolekcje



Interfejs Collection, oprócz tego że stoi na szczycie hierarchii jest przy okazji dostawcą większości metod, z których korzystają dziedziczące interfejsy. Do metod tych należą między innymi: add (dodawanie do kolekcji), remove (usuwanie z kolekcji), clear (czyszczenie kolekcji), size (zwracanie rozmiaru kolekcji).

List - Najczęściej używany rodzaj kolekcji. Charakteryzuje się tym, że jej elementy posiadają określoną kolejność. Dodatkowo są one indeksowane, dzięki czemu można pobrać obiekt znajdujący się na konkretnej pozycji (indeksie) w liście. Na przykład możemy użyć metody get(2), gdzie 2 to numer pozycji, którą chcemy pobrać. Dostaniemy wtedy trzeci element listy (pozycje są numerowane od 0). Interfejs jest implementowany przez następujące klasy: ArrayList, LinkedList, Vector (ta klasa jest jeszcze rozszerzana przez Stack, który nie implementuje bezpośrednio interfejsu List)

Queue - Najważniejszą cechą tej kolejki jest zachowanie kolejności według zasady FIFO (First In - First Out). Oznacza to, że pierwszy element wprowadzony do kolejki będzie z niej również pobierany jako pierwszy. Interfejs jest implementowany przez: PriorityQueue

Deque (Double ended queue) - Kolejki te, zgodnie z nazwą, wspierają zarówno FIFO jak i LIFO (Last In - First Out). To drugie oznacza, że ostatni element wprowadzony do kolejki będzie z niej pobierany jako pierwszy. W tym miejscu warto sobie wyobrazić stos, na którym układamy kolejne elementy, by później zdejmować je począwszy od tego na samej górze i potem dalej w dół. Interfejs jest implementowany przez: ArrayDeque, LinkedList (ta klasa implementuje zarówno interfejs List jak i Deque)

Set - Najważniejszą cechą tego rodzaju kolekcji jest unikalność jej elementów. Porównanie obiektów następuje przez wykorzystanie metody equals. W zbiorze nigdy nie będą istniały dwa obiekty (ani więcej), dla których metoda ta zwraca wartość true. Sety nie posiadają numerowanych pozycji w postaci indeksów. Zatem nie odwołamy się tutaj do konkretnej pozycji celem pobrania obiektu. Interfejs jest implementowany przez klasy: HashSet, LinkedHashSet (rozszerza klasę HashSet, ale implementuje też bezpośrednio interfejs Set),

SortedSet. - Zbiory posortowane mają dokładnie te same cechy co "zwykłe" zbiory, z tym że dodatkowo zapewniają one kolejność elementów. Sortowanie odbywa się na podstawie wykorzystania specjalnej metody, o której jeszcze nie mówiliśmy, ale która pojawi się w niedługim czasie w naszym kursie. Interfejs jest implementowany przez: TreeSet

LISTY

Listy są dostarczane w ramach kolekcji za pośrednictwem interfejsu List z pakietu java.util. Listę możemy zasadniczo zdefiniować jako bardziej elastyczną wersję tablicy. Jest to jednak pewne uproszczenie, dlatego też przyjrzyjmy się teraz dokładnie jakie są cechy charakterystyczne list:

- Elementy listy mają określoną kolejność.
- Dozwolone są duplikaty elementów.
- Elementy można umieścić na liście w określonej pozycji.
- Element znajdujący się w określonej pozycji można prosto pobrać z listy.
- Element w określonej pozycji możemy łatwo podmienić na inny.
- Listy możemy łatwo sortować.
- Nie musimy deklarować na początku rozmiaru listy (choć ogólnie warto to robić, jeśli jesteśmy w stanie przewidzieć jej rozmiar).

TWORZENIE LISTY

Dobrą praktyką jest zadeklarowanie instancji listy z parametrem typu, na przykład:

```
List<Object> listAnything = new ArrayList<Object>();
List<Integer> listNumbers = new ArrayList<Integer>();
List<String> linkedWords = new LinkedList<String>();
```

Od Javy 7 możemy usunąć parametr typu po prawej stronie, co upraszcza zapis do następującej postaci:

```
List<Integer> listNumbers = new ArrayList<>();
List<String> linkedWords = new LinkedList<>();
```

Od Javy 9 możemy utworzyć kolekcję z ustalonego zestawu elementów:

```
List listNumbers = List.of(1, 2, 3, 4, 5, 6);
```

Od Javy 10 można jeszcze bardziej skrócić tworzenie kolekcji, używając słowa var:

```
var items = new ArrayList<Item>();
```

PODSTAWOWE OPERACJE NA LISTACH

Podstawowymi operacjami, jakie możemy wykonywać na listach, jest dodawanie, pobieranie, aktualizowanie oraz usuwanie elementów. Operacje te są realizowane przez następujące metody:

- add(<obiekt>) - umożliwia ona dodanie elementu do listy. Co ważne - wymagane jest dodanie elementów tego samego typu (lub podtypu) co parametr typu zadeklarowanego przez listę.

```
package PO_UR.Lab09;

import java.util.ArrayList;
import java.util.List;
```

```
public class ListyExample {
    public static void main(String[] args) {

        List<String> listStrings = new ArrayList<String>();
        // Poniższe elementy zostaną dodane do listy - wszystkie są typu
String:
        listStrings.add("One");
        listStrings.add("Two");
        listStrings.add("Three");
        // Poniższy element nie zostanie dodany do listy - kompilator
zgłosi błąd:
        listStrings.add(123);
    }
}
```

Dodawanie elementów podtypów zadeklarowanego typu:

```
List<Number> linkedNumbers = new LinkedList<>();
linkedNumbers.add(new Integer(123));
linkedNumbers.add(new Float(3.1415));
linkedNumbers.add(new Double(299.988));
linkedNumbers.add(new Long(67000));
```

Pozycje w liście - podobnie jak w przypadku tablic - są numerowane od zera. Dodawanie następnego elementu powoduje umieszczenie go w kolejnym wolnym indeksie (0, 1, 2, 3...itd.).

- get(<indeks>) - umożliwia ona pobranie elementu z listy poprzez podanie indeksu, na którym znajduje się ten element.

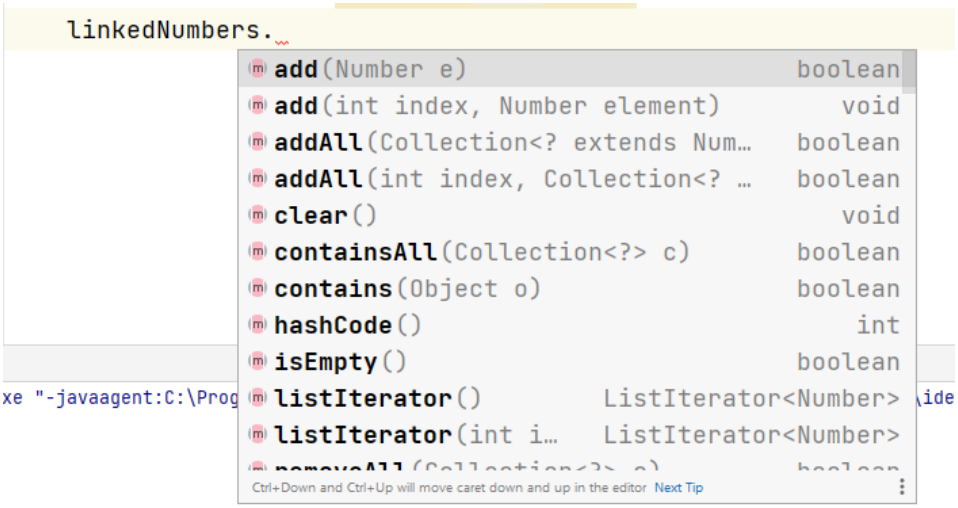
```
String element = listStrings.get(1);
Number number = linkedNumbers.get(3);
```

- set(<indeks>,<obiekt>) - metoda aktualizuje element w danej pozycji listy, a dokładniej mówiąc zastępuje ten element.

```
listStrings.set(3, "Hello Java");
```

- remove(<indeks>) -metoda usuwa element w danej pozycji listy. Innymi słowy, element znajdujący się na danym indeksie jest całkowicie usuwany z listy.

Pozostałe metody umożliwiające pracę z listami poznamy z biegiem czasy pracy na listach, metody dostępne są po operatorze kropki:



PRZEGLĄDANIE ZAWARTOŚCI LISTY

Listy implementują interfejs Iterable, co umożliwia przeglądanie ich element po elemencie. W tym celu wykorzystać można interfejs Iterator. Jego działanie polega na przeglądaniu listy, dopóki po danym elemencie występuje kolejny element. Pobranie bieżącego elementu i przejście do następnego wykonywane jest za pomocą metody next.

```
package PO_UR.Lab09;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

public class ListyExample {
    public static void main(String[] args) {
        List<Number> someNumbers = new ArrayList<>();
        someNumbers.add(new Integer(123));
        someNumbers.add(new Float(3.1415));
        someNumbers.add(new Double(299.988));
        someNumbers.add(new Long(67000));

        Iterator<Number> someNumbersIterator = someNumbers.iterator();

        while(someNumbersIterator.hasNext()) {
            System.out.println(someNumbersIterator.next());
        }
    }
}
```

Wykorzystanie pętli foreach

```
package PO_UR.Lab09;

import java.util.ArrayList;
import java.util.List;

public class ListyExample {
    public static void main(String[] args) {

        List<Number> someNumbers = new ArrayList<>();
        someNumbers.add(new Integer(123));
        someNumbers.add(new Float(3.1415));
        someNumbers.add(new Double(299.988));
        someNumbers.add(new Long(67000));

        for (Number number : someNumbers) {
            System.out.println(number);
        }
    }
}
```

SETY

Kolejną kolekcją w Javie są Sety, inaczej mówiąc zbiory. Dostarczane są one za pośrednictwem interfejsu Set z pakietu java.util. Sety wyglądają z wierzchu trochę podobnie do list, jednak ich implementacja i podejście do danych jest zupełnie inne. Przyjrzyjmy się teraz najważniejszym cechom zbiorów:

- Nie są dozwolone duplikaty elementów.

- Do elementów w secie nie odwołujemy się po pozycji, gdyż nie wiadomo na jakiej pozycji znajduje się dany element.
- Mogą być sortowalne lub nie - zależy od konkretnej implementacji interfejsu.

Na początku można zadać sobie pytanie dotyczące użyteczności setów, których konstrukcja wydaje się być bardziej zagmatwana od list i do tego ma swoje ograniczenia. Mimo tego sety są bardzo często używane i nie można ich pominąć w trakcie nauki. Dzieje się tak ze względu na ich specjalną właściwość:

- Unikalność elementów
- W secie nie mogą istnieć dokładnie dwa takie same obiekty. Zawsze będzie przechowywany tylko jeden. Zatem jeśli mamy ciąg liczb: 7, 2, 9, 1, 5, 7, 6, 3, 9 i taki ciąg wpiszemy do seta, to odczytując wszystkie elementy zbioru otrzymamy tylko pojedyncze wartości: 7, 2, 9, 1, 5, 6, 3. W ten sposób można w łatwy sposób odfiltrować duplikaty. Wystarczy wprowadzić je do seta. Wyciągniemy z niego jedynie niezduplikowane elementy.

WERYFIKACJA UNIKALNOŚCI - KONTRAKT EQUALS/HASHCODE

To czy dany element znajduje się w zbiorze jest weryfikowane przy pomocy metod hashCode i equals. Metody te (występujące w każdym obiekcie) wiąże istotny kontrakt (opisujący relację między nimi). Polega on na tym, że jeśli dwa obiekty są równe według metody equals to oznacza, że muszą mieć te same "haszkody" (metoda hashCode zwraca dla nich tę samą wartość). Natomiast jeśli obiekty mają równe "haszkody" niekoniecznie muszą być równe według porównania metodą equals.

TWORZENIE ZBIORU

Dobrą praktyką jest zadeklarowanie instancji zbioru z parametrem typu, na przykład:

```
Set<Object> listAnything = new HashSet<Object>();
Set<Integer> listNumbers = new HashSet<Integer>();
Set<String> linkedWords = new LinkedHashSet<String>();
Set<String> sortedWords = new TreeSet<String>();
```

Podobnie jak to miało miejsce w przypadku list, tak samo tutaj możemy określić jakiego typu obiekty mogą być przechowywane w danym zbiorze. Dzięki temu już na etapie kompilacji jesteśmy w stanie się zorientować, czy nie popełniliśmy błędu - na przykład przekazując obiekt klasy String do zbioru, w którym możemy przechowywać tylko obiekty klasy Integer. W takiej sytuacji dowiemy się o tym od razu, gdyż po prostu otrzymamy błąd kompilacji.

PODSTAWOWE OPERACJE NA ZBIORACH

- add(<obiekt>) - umożliwia ona dodanie elementu do seta. Co ważne - wymagane jest dodanie elementów tego samego typu (lub podtypu) co parametr typu zadeklarowanego przez seta.

```
Set<String> strings = new HashSet<String>();
// Poniższe elementy zostaną dodane do seta - wszystkie są typu String:
strings.add("One");
strings.add("Two");
strings.add("Three");
// Poniższy element nie zostanie dodany do seta - kompilator zgłosi błąd:
strings.add(123);
```

DODAWANIE ELEMENTÓW PODTYPÓW ZADEKLAROWANEGO TYPU:

```
Set<Number> linkedNumbers = new LinkedHashSet<>();
linkedNumbers.add(new Integer(123));
```

```
linkedNumbers.add(new Float(3.1415));
linkedNumbers.add(new Double(299.988));
linkedNumbers.add(new Long(67000));
```

Pamiętajmy, że w przypadku setów nie możemy mówić o indeksie elementu. W przypadku LinkedHashSet mamy jedynie zapewnioną kolejność, czyli podczas przeglądania zbioru otrzymamy elementy zgodne z kolejnością ich dodawania.

- remove(<obiekt>) - metoda usuwa element ze zbioru. `strings.remove("One");`

PRZEGLĄDANIE ZAWARTOŚCI SETA

Zbiory implementują interfejs Iterable, co umożliwia przeglądanie ich element po elemencie. W tym celu wykorzystujemy interfejs Iterator. Jego działanie polega na przeglądaniu kolekcji dopóki po danym elemencie występuje kolejny element. Pobranie bieżącego elementu i przejście do następnego wykonywane jest za pomocą metody next.

```
Set<Number> someNumbers = new HashSet<>();
someNumbers.add(new Integer(123));
someNumbers.add(new Float(3.1415));
someNumbers.add(new Double(299.988));
someNumbers.add(new Long(67000));

Iterator<Number> someNumbersIterator = someNumbers.iterator();

while(someNumbersIterator.hasNext()) {
    System.out.println(someNumbersIterator.next());
}
```

foreach

```
Set<Number> someNumbers = new HashSet<>();
someNumbers.add(new Integer(123));
someNumbers.add(new Float(3.1415));
someNumbers.add(new Double(299.988));
someNumbers.add(new Long(67000));

for (Number number : someNumbers) {
    System.out.println(number);
}
```

MODYFIKOWANIE OBIEKTÓW W SETACH

Omawiając listy wspomnieliśmy o metodzie set, która pozwalała nam na aktualizację obiektu na danej pozycji. W przypadku setów nie mamy indeksów i nie mamy metody set. Co nam zatem pozostaje? Otóż możemy iterować po kolejnych obiektach seta i porównywać je po wybranym polu w celu znalezienia właściwego obiektu, a gdy już to zrobimy możemy wtedy zaktualizować jego dane. Inną opcją jest usunięcie obiektu z seta metodą remove i dodanie go na nowo w zaktualizowanej wersji za pomocą metody add.

Zadania do samodzielnego rozwiązania:

Zadanie 1.

Napisz program, który będzie pobierał od użytkownika imiona. Program powinien pozwolić użytkownikowi na wprowadzenie dowolnej liczby imion (wprowadzenie „-” jako imienia przerwie wprowadzanie). Na zakończenie wypisz liczbę unikalnych imion.

Zadanie 2.

Napisz program, który będzie pobierał od użytkownika imiona par dopóki nie wprowadzi imienia „-”, następnie poproś użytkownika o podanie jednego z wcześniej wprowadzonych imion i wyświetl imię odpowiadającego mu partnera.

Zadanie 3.

Napisz klasę przechowującą informacje o uczestnikach wydarzenia (ID, imię oraz jego wiek). Zaimplementować metodę toString(), aby wyświetlać informację o uczestniku oraz metody equals() oraz hashCode() (metody do porównywania obiektów). Do przechowywania uczestników należy użyć listy. Ponadto zaproponować metodę pozwalającą na filtrowanie osób niepełnoletnich. Zaproponować rozwiązanie z użyciem LinkedList oraz ArrayList.

Zadanie 4.

Utwórz dowolną klasę, a potem zainicjalizowaną tablicę obiektów tej klasy. Zawartością tablicy wypełnij listę List. Wyłuskaj z niej fragment listy metodą subList(), a następnie usuń tę podlistę z oryginalnej listy.

Zadanie 5.

Utwórz i zapełnij listę List<Integer>. Utwórz drugą listę List<Integer>. Użyj ListIterator do przejrzenia elementów pierwszej listy i wstawienia ich do listy drugiej, ale w odwrotnej kolejności.