Kolekcje - wprowadzenie

Czego się dowiesz

- Czym są kolekcje,
- jaki jest podział kolekcji w Javie,
- jak wybrać odpowiedni typ kolekcji.

Wstęp

Programy wykorzystujące tablice do przechowywania większych porcji danych posiadają jeden główny problem - tablice mają z góry ustalony rozmiar. W zadaniu dotyczącym wykorzystania klasy Arrays zaimplementowaliśmy mechanizm automatycznego rozszerzania tablicy używając metody System.arraycopy(), jednak nie jest to nadal rozwiązanie bardzo uniwersalne, a dodatkowo mało czytelne i trzeba byłoby je powtarzać niemal w każdej aplikacji.

Już od samego początku istnienia Javy dostępna była klasa Vector, która w pewien sposób rozwiązywała ten problem. Można powiedzieć, że była to pierwsza "dynamiczna tablica". W wersji Javy 1.2 wprowadzono jednak dużą aktualizację związaną m.in. z dodaniem gotowego i uniwersalnego zbioru kilku rodzajów kolekcji o nazwie **Collections Framework**. Znajdziemy w nim kilka bardzo użytecznych struktur danych, które zdejmują z programisty obowiązek martwienia się o to jak zaimplementować "rozszerzającą się tablicę”.

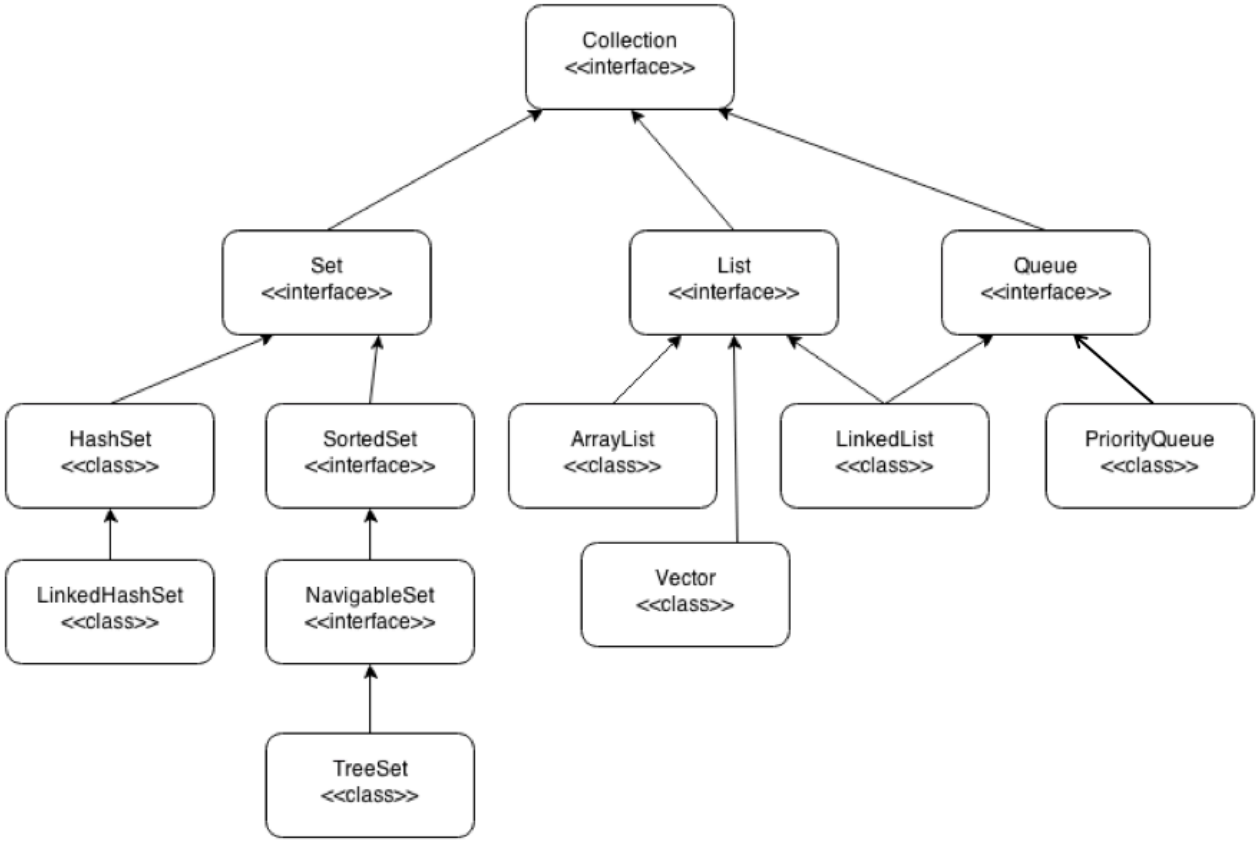
Podstawowy podział kolekcji dostępnych domyślnie w Javie to:

- **Listy** - uporządkowane kolekcje obiektów, które są numerowane za pomocą **indeksów**,

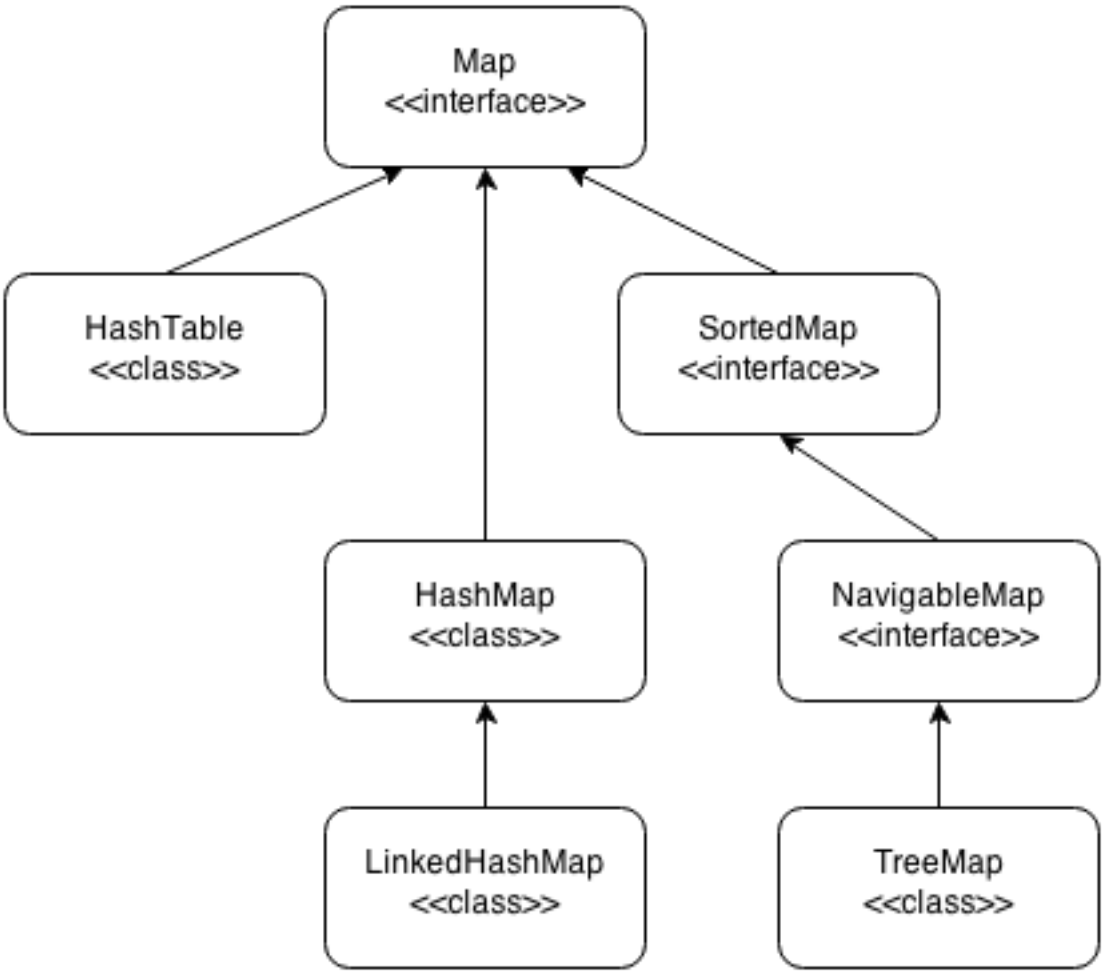
- **Zbiory** - zgodnie z matematyczną definicją zbioru są to kolekcje **unikalnych** obiektów,
- **Mapy** - kolekcje przechowujące dane na zasadzie **klucz-wartość**, gdzie klucz jest unikalnym identyfikatorem pozwalającym wyszukać wartość,
- **Kolejki** - można powiedzieć, że to specyficzny typ listy. Dobrze jest sobie ją wyobrazić jako kolejkę w sklepie - jeżeli stanąłeś przy kasie jako pierwszy, to pierwszy zostaniesz obsłużony.
-

Poszczególnym z powyższych kolekcji w Javie odpowiadają interfejsy List, Set, Map oraz Queue. Wszystkie z nich od wersji Javy 1.5 są generyczne, więc możemy określić jaki typ obiektów będą przechowywać. Z powodu generyczności nie mogą one przechowywać typów prostych, ale dzięki autoboxingowi i unboxingowi nie jest to problemem.

Poniżej przedstawione jest drzewo podziału interfejsów i klas kolekcyjnych w collections framework.



Mapy także są składową collections framework, jednak ze względu na swój specyficzny sposób działania (klucz-wartość), oraz mocno odmienny zbiór metod, nie są one pochodną nadrzędnego interfejsu Collection.



W kolejnych lekcjach omówimy najważniejsze z wyżej wymienionych kolekcji i pokażemy ich praktyczne zastosowanie.

Listy (ArrayList I LinkedList)

Czego się dowiesz

- Czym są listy,
- jak działa lista tablicowa (ArrayList),
- jak działa lista wiązana (LinkedList),
- kiedy wykorzystać konkretny typ listy.

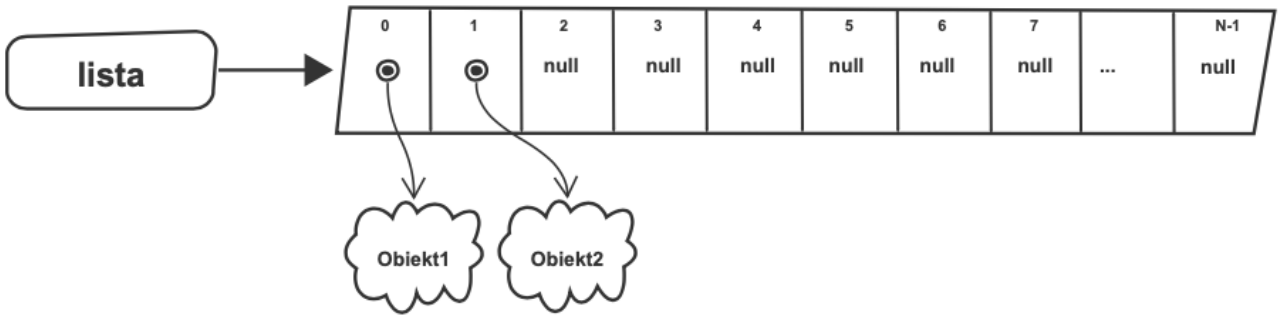
Wstęp

Listy pozwalają na przechowywanie informacji w uporządkowanej i indeksowanej kolejności. Dzięki tym cechom, a także faktowi, że ich pojemność jest niemal nieograniczona (ograniczeniem jest wielkość wartości int, oraz ewentualna ilość pamięci maszyny wirtualnej), są one świetnym sposobem na zastąpienie tablic w większości przypadków. Pamiętaj jednak, że mimo wszystko tablic nie należy w tym momencie przekreślać. W aplikacjach wymagających dużej wydajności i szybkiego dostępu do danych, tablice nadal mogą okazać się najlepszym wyborem.

ArrayList (listy tablicowe)

Lista tablicowa w swojej wewnętrznej reprezentacji wykorzystuje znane ci już doskonale tablice. Różnica polega na tym, że jako użytkownik klasy **ArrayList** nie musisz się przejmować tym wszystkim co dzieje się w środku. Możemy w konstruktorze klasy zapewnić pewien domyślny rozmiar wewnętrznej tablicy lub pozostawić go domyślnym, który wynosi 10. W uproszczeniu lista tablicowa wygląda w taki sposób:

Chociaż ze względu na to, że kolekcje w Javie mogą przechowywać jedynie obiekty, a nie typy proste, to w rzeczywistości lista przechowuje tylko referencje wskazujące na obiekty lub wartości null, podobnie jak to było przy tablicach, np.:



Dzięki temu, że wewnątrz klasy ArrayList operujemy na tablicy, to dostęp do danych jest bardzo szybki. Podobnie szybkie jest wstawianie elementu na koniec takiej listy, czyli pierwsze wolne miejsce.

Usuwanie lub wstawianie elementów w środku listy wiąże się z koniecznością przesuwania sporej ilości elementów. Przykładowo jeśli lista przechowuje 100 obiektów, a my usuniemy element na indeksie 2, to wszystkie elementy, które leżą "na prawo" od niego, muszą być przesunięte o 1 miejsce w lewo, co jest dosyć kosztowne.

Ponieważ klasy kolekcji są w Javie generyczne, to deklaracja ArrayListy będzie wyglądała następująco:

```
ArrayList<Typ_Obiektowy> nazwaListy = new ArrayList<>(opcjonalny_domyślny_rozmiar);
```

W wersjach Javy starszych niż 8 musielibyśmy w nawiasach trójkątnych przy tworzeniu obiektu również podać Typ_Obiektu, jednak od Javy 8 mamy do dyspozycji operator diamentu, który skraca ten zapis. Rozmiar tablicy powinniśmy zadeklarować na inny niż domyślny przede wszystkim w sytuacji gdy wiemy, że na pewno będzie ona przechowywała dużo więcej obiektów - unikniemy wtedy potencjalnego spowolnienia związanego z rozszerzaniem wewnętrznej tablicy.

Dużą różnicą w porównaniu do tablicy jest też fakt, że lista tablicowa nie jest traktowana w żaden specjalny sposób, jest po prostu zwykłym obiektem klasy ArrayList. Jeżeli więc chcemy dodać jakiś element lub się do niego odwołać, musimy korzystać z metod, a nie korzystać z jakiejś nowej notacji z użyciem kwadratowych nawiasów. Najważniejsze z metod w tej klasie to:

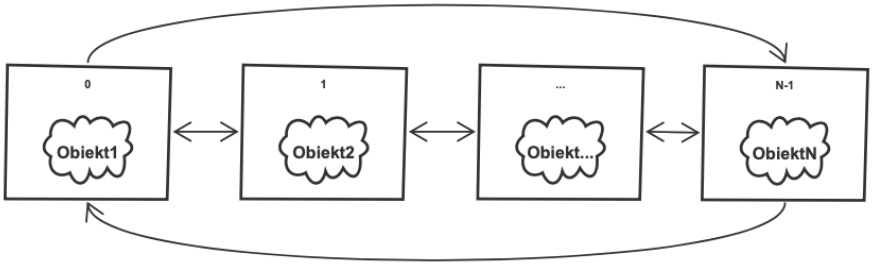
- **add(E e)** - dodaje element zgodny z zadeklarowanym typem generycznym do kolekcji. Pozwala na dodawanie wartości null oraz powtarzających się elementów,
- **addAll(Collection c)** - dodaje wszystkie elementy z innej kolekcji do listy (kopie referencji),
- **clear()** - usuwa wszystkie elementy z listy,
- **get(int index)** - zwraca element pod podanym indeksem. Podobnie jak przy tablicach indeksowanie jest od 0. Metoda może generować wyjątek ArrayIndexOutOfBoundsException,
- **remove(int index)** - usuwa element z indeksu podanego jako parametr,
- **remove(Object o)** - usuwa pierwsze wystąpienie obiektu, dla którego metoda equals() zwróci true,
- **sort(Comparator c)** - sortuje listę zgodnie z przekazanym komparatorem,
- **size()** - zwraca rozmiar listy. Działanie analogiczne do właściwości length tablicy,
- (pełną listę metod znajdziesz w dokumentacji)

Lists.java

```
import java.util.ArrayList;

class Lists {
    public static void main(String[] args) {
        ArrayList<Integer> ints = new ArrayList<>();

        for (int i = 0; i < 100; i++) {
            ints.add(i); // obiekty Integer, nie int!
        }

        System.out.println("ints.get(50) = " + ints.get(50));
        System.out.println("Ilość elementów na liście: " + ints.size());
        System.out.println("Czyszczę tablicę: ");
        ints.clear();
        System.out.println("Ilość elementów na liście: " + ints.size());
    }
}
```

Listy wiązane (LinkedList)

Listy wiązane w odróżnieniu od list tablicowych do wewnętrznej reprezentacji danych nie wykorzystują tablic, lecz obiekty dodatkowej klasy wewnętrznej, które opakowują, każdy dodawany obiekt i tworzą powiązanie z poprzednim elementem.



W Javie klasa **LinkedList** jest zaimplementowana jako lista podwójnie wiązana, tzn., że każdy jej element przechowuje referencję do elementu poprzedzającego oraz do swojego następnika oraz przechowywaną wartość (referencję na obiekt). Jeżeli mamy więc listę przechowującą 100 obiektów, to aby pobrać obiekt zapisany na 49 miejscu, musimy przejść po elementach 0>1>2>3>...>49, a w przypadku, gdy obiekt jest bliżej końca, konieczne jest przejście ścieżki odwrotnej, np. 99>98>97>...>65**.** Z tego powodu operacje wyszukiwania, czy też generalnie odczytu danych z środka listy wiązanej nie są zbyt efektywne. Szybciej niż w przypadku ArrayListy działa natomiast wstawianie i usuwanie elementów z listy, ponieważ nie ma tutaj konieczności przesuwania całego zbioru danych - wystarczy zaktualizować następniki i poprzedniki odpowiednich elementów (dwie referencje).

W LinkedList oprócz metod wymienionych przy ArrayList, znajdziemy też zdefiniowane dodatkowe metody, które wynikają z tego w jaki sposób przechowywane są w niej elementy:

- **addFirst(E e) / removeFirst()** - dodanie lub usunięcie elementu z początku listy,
- **addLast(E e) / removeLast()** - dodanie lub usunięcie elementu z końca listy,
- (pełną listę metod znajdziesz w dokumentacji).

Deklaracja, inicjalizacja i operacje na liście tego typu są analogiczne jak na obiekcie klasy ArrayList.
W poprzednim przykładzie wystarczy podmienić linijkę z deklaracją listy na poniższą, a reszta kodu nadal będzie działać:

```
LinkedList<Integer> ints = new LinkedList<>();
```

Listy i Java 9

Jeżeli chcemy utworzyć niewielką listę i znamy elementy, które będą do niej wstawione, to przed Javą 9 najwygodniejszym sposobem było wykorzystanie metody Arrays.asList(), np.:

```
import java.util.Arrays;
import java.util.List;

class ArrTest {
    public static void main(String[] args) {
        Integer[] ints = { 5, 10, 15 };
        List<Integer> lista = Arrays.asList(ints);
        for (Integer num : lista) {
            System.out.println(num);
        }
    }
}
```

rozwiązanie takie jest ok, jednak jak na tak prostą czynność jak utworzenie listy o kilku elementach wymaga to pisania sporej ilości niepotrzebnego kodu i musimy tworzyć tablicę, która nie jest nam tutaj potrzebna. Java 9 wprowadza tzw. **collections factory methods**, czyli specjalne metody, przeznaczone przede wszystkim do tworzenia niewielkich list, gdy znamy elementy, które mają się w niej znaleźć. Od teraz w celu utworzenia listy można zapisać:

```
import java.util.List;

class ArrTest {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(5, 10, 15);
        for (Integer num : numbers) {
            System.out.println(num);
        }
    }
}
```

Co ciekawe istnieje wiele przeciążonych wersji metody List.of() dla 1, 2, 3, ..., 10 elementów. Istnieje też specjalna wersja tej metody przyjmująca dowolną liczbę argumentów, ale nie powinna ona być nam zbyt często potrzebna. Rozwiązanie takie jest podyktowane kwestią wydajności i przy niewielkich tablicach pozwala uniknąć tworzenia dodatkowej tablicy.

Bardzo ważną kwestią przy tworzeniu list z użyciem powyższej metody jest to, że lista taka jest niemodyfikowalna, czyli nie możemy do niej ani dodawać, ani usuwać z niej elementów. Jeśli spróbujemy to zrobić to otrzymamy wyjątek UnsupportedOperationException.

Listy i inne kolekcje a polimorfizm

Ponieważ zarówno klasy ArrayList jak i LinkedList implementują wspólny interfejs List, to wiele z ich metod jest współdzielonych. Dzięki temu w wielu sytuacjach lepiej jest pisać programy operujące na jak najbardziej abstrakcyjnych typach kolekcji. Przykładowo jeśli chcemy napisać metodę, która wyświetla wszystkie elementy listy, to lepiej jest zadeklarować, że przyjmuje ona jako parametr obiekt typu List, a nie LinkedList, czy ArrayList, bo dzięki temu będziemy mogli do niej przekazać zarówno te pierwsze jak i drugie:

PolymorphicLists.java

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

class PolymorphicLists {
    public static void main(String[] args) {
        LinkedList<Integer> ints1 = new LinkedList<>();
        ArrayList<Integer> ints2 = new ArrayList<>();

        for (int i = 0; i < 100; i++) {
            ints1.add(i);
            ints2.add(i);
        }

        printList(ints1);
        printList(ints2);

    }
```

```
    static void printList(List<Integer> list) {
        for (Integer n : list) {
            System.out.println(n);
        }
    }
}
```

Metoda printList(), dzięki przyjmowaniu ogólniejszego typu kolekcji jest bardziej uniwersalna. Jeśli nie wykorzystujesz metod specyficznych dla danej implementacji, np. removeLast() z klasy LinkedList, to rozważ także posługiwanie się referencjami ogólniejszego typu dla zwykłych zmiennych, czyli, np:

```
List<Integer> ints1 = new LinkedList<>();
List<Integer> ints2 = new ArrayList<>();
```

Jeśli jednak potrzebujesz metodę z konkretnego typu, np. wspomniana removeLast() to definiuj od razu kolekcję konkretnego typu, aby uniknąć rzutowania. Np.:

```
List<Integer> link = new LinkedList<>();
link.add(5);
link.add(10);
link.add(15);

link.removeLast(); // błąd
```

W powyższym kodzie nie mamy dostępu do metody removeLast(), ponieważ zadeklarowaliśmy zmienną jako ogólniejszy typ List. Deklarując zmienną jako LinkedList problem nie występuje:

```
LinkedList<Integer> link = new LinkedList<>();
link.add(5);
link.add(10);
link.add(15);

link.removeLast(); // ok
```

Powyższe rady dotyczą wszystkich typów kolekcji, nie tylko list. Ogólna zasada brzmi: definiuj zmienne jako najogólniejszy możliwy typ, który nie powoduje utraty funkcjonalności i nie zmusza cię do rzutowania.