Java - kolekcje i operacje wejścia/wyjścia

Kolekcje

Kolekcje to klasy służące do przechowywania obiektów. Najczęściej spotykane rodzaje kolekcji to:

- listy,
- mapy,
- zbiory.

Listy

Poznaliśmy już wcześniej najprostszą kolekcję - listę. Lista pozwala na:

- przechowywanie obiektów dowolnej klasy,
- dynamiczną zmianę rozmiaru (dodawanie i usuwanie obiektów),
- iterację po obiektach w takiej kolejności, w jakiej zostały dodane do listy.

```
List<String> nazwiska = new ArrayList<>();
nazwiska.add("Kowalski");
nazwiska.add("Nowak");
```

- W powyższym przykładzie:
- `List` to nazwa interfejsu, który implementują wszystkie listy;
- `ArrayList` to nazwa konkretnej klasy, której użyliśmy do przechowywania obiektów; klasa ta implementuje interfejs `List`;
- `<String>` oznacza, że na liście przechowywane będą obiekty klasy `String`.
- Po liście możemy iterować pętlą for-each lub też pobrać dowolny jej element metodą `get()`:

```
for(String nazwisko : nazwiska) {
    System.out.println(nazwisko);
}

for(int i=0; i<nazwiska.size(); i++) {
    String nazwisko = nazwiska.get(i);
    System.out.println(nazwisko);
}
```

Mapy

Mapa przechowuje pary klucz-wartość. Bywa też nazywana słownikiem lub tablicą haszującą. Przykład użycia:

```
Map<String, String> dniTygodnia = new HashMap<>();
dniTygodnia.put("pn", "poniedziałek");
```

```
dniTygodnia.put("wt", "wtorek");
dniTygodnia.put("śr", "środa");
dniTygodnia.put("czw", "czwartek");
```

- W powyższym przykładzie:
- `Map` to nazwa interfejsu, implementowanego przez wszystkie mapy;
- `HashMap` to nazwa konkretnej klasy, implementującej interfejs `Map`; moglibyśmy zamiast niej użyć np. `TreeMap`;
- `<String, String>` oznacza, że mapa kojarzyć będzie klucz typu `String` z wartością typu `String`.
- Mapa `dniTygodnia` przechowywać będzie pełne nazwy dni tygodnia (wartość), skojarzone ze skrótem (klucz). Aby pobrać pełną nazwę dla skrótu "pn":

  ```
  String dzien = dniTygodnia.get("pn");
  System.out.println("Skrót 'pn' oznacza: " + dzien);
  ```

- Iteracja po mapie jest nieco trudniejsza, niż w przypadku listy. Najprościej pobrać zbiór kluczy z mapy metodą `keySet()`, a następnie iterować po tym zbiorze:

  ```
  for(String skrot : dniTygodnia.keySet()) {
      String dzien = dniTygodnia.get(skrot);
      System.out.println(skrot + " - " + dzien);
  }
  ```

Mapa nie gwarantuje kolejności iteracji - nie wiadomo z góry, w jakiej kolejności zostaną zwrócone elementy.

- Mapa może przechowywać obiekty dowolnych klas, np. możemy zmapować pesel na osobę:

  ```
  Map<String, Person> osoby = new HashMap<>();
  osoby.put("90010112345", new Person("Jan", "Kowalski"));
  ```

  albo numer grupy na listę studentów:

  ```
  Map<Integer, List<Student>> grupy = new HashMap<>();

  List<Student> grupa32 = new ArrayList<>();
  grupa32.add(new Student("Jan", "Kowalski"));
  grupa32.add(new Student("Zenon", "Nowak"));

  grupy.put(32, grupa32);
  ```

Zbiory

Zbiory, podobnie jak listy, przechowują dowolną liczbę obiektów. Obiekty dodajemy metodą `add()`:

```
Set<String> zbiorDni = new HashSet<>();
zbiorDni.add("poniedziałek");
zbiorDni.add("wtorek");

System.out.println(zbiorDni);
```

- Zbiór nie może zawierać powtarzających się elementów - dodanie do zbioru "wtorek" dwukrotnie nie będzie miało żadnego efektu:

  ```
  zbiorDni.add("wtorek");
  zbiorDni.add("wtorek");
  zbiorDni.add("wtorek");
  System.out.println(zbiorDni);
  ```

- W zbiorze nie ma informacji o indeksie obiektu, zbiór nie zachowuje kolejności obiektów. Można po nim iterować pętlą for-each, nie mamy przy tym żadnej gwarancji, w jakiej kolejności zostaną zwrócone elementy:

  ```
  for(String dzien : zbiorDni) {
      System.out.println(dzien);
  }
  ```

Zadanie 1

- Dodaj do zbioru wszystkie dni tygodnia.
- W ostatnim przykładzie użyta została klasa `HashSet` - co się stanie, jeśli zmienimy ją na `TreeSet`? Zmień tylko linijkę z deklaracją - zamiast:

  ```
  Set<String> zbiorDni = new HashSet<>();
  ```

  wstaw:

  ```
  Set<String> zbiorDni = new TreeSet<>();
  ```

Sprawdź efekt. Zastanów się:

- czym się różni `HashSet` od `TreeSet`?
- której klasy użyć w jakiej sytuacji?
- czy były wymagane jakiekolwiek zmiany w kodzie poza miejscem tworzenia obiektu (`new HashSet<>()` / `new TreeSet<>()`) ?

Zadanie 2

- Wykorzystaj kod z poprzednich zajęć z klasami `Person`, `Student`, `Teacher`.

- Przygotuj mapę, w której nazwa przedmiotu mapowana będzie na nauczyciela prowadzącego. Dodaj kilku nauczycieli do mapy, wyciągnij z mapy nauczyciela prowadzącego "programowanie obiektowe".
- Przygotuj kilka list zawierających grupy studentów.
- Przygotuj mapę, w której numer grupy będzie mapowany na listę studentów. Dodaj do niej wcześniej stworzone listy. Wyciągnij i wyświetl listę studentów grupy nr 32.

Odczyt plików

Standardowa biblioteka Javy zawiera wiele (zbyt wiele...) różnych klas pozwalających na odczyt i zapis do strumieni. Taki strumień może reprezentować plik na dysku, połączenie sieciowe, albo np. tablicę bajtów w pamięci. W kazdym przypadku interfejs pozostaje bardzo podobny.

- Pliki tekstowe najprościej odczytywać za pomocą klasy `BufferedReader`:

  ```
  BufferedReader in = new BufferedReader(new FileReader("plik.txt"));
  ```

- Odczytanie linii tekstu:

  ```
  String s = in.readLine();
  ```

  Jeśli osiągniemy koniec pliku, metoda `readLine()` zwróci wartość `null`.

- Wszystkie operacje wejścia/wyjścia mogą spowodować wyjątek `IOExcpetion`. Musimy ten wyjątek obsłużyć w bloku `try...catch`:

  ```
  try {
      ... tu będą operacje na pliku
  } catch (IOException ex) {
      ex.printStackTrace();
  }
  ```

- Aby strumień został automatycznie zamknięty po zakończeniu odczytu/zapisu, użyjemy konstrukcji *try with resources*:

  ```
  try(BufferedReader in = new BufferedReader(new FileReader("plik.txt"))) {

      String s = in.readLine();



  } catch (IOException ex) {
      ex.printStackTrace();
  }
  ```

  Dzięki temu Java automatycznie zamknie strumień `in` po wyjściu z bloku `try` - bez względu na to, czy kod wykona się poprawnie, czy też zostanie rzucony wyjątek.

- Umieść w katalogu projektu plik o nazwie `plik.txt` z kilkoma linijkami tekstu i wypróbuj odczyt pliku. Oto kompletny kod odczytujacy plik linia po linii i wypisujący zawartość na ekranie:

```
try(BufferedReader in = new BufferedReader(new FileReader("plik.txt"))) {

    String s = in.readLine();

    while(s!=null) {
        System.out.println(s);

        s = in.readLine();
    }
} catch (IOException ex) {
    ex.printStackTrace();
}
```

Zadanie 3

- Pobierz plik zawierający listę produktów z identyfikatorami, nazwami, cenami i kategoriami.
- Stwórz klasę `Product` z polami `id`, `name`, `price`, `category`.
- Odczytaj listę produktów z pliku. Dla każdej odczytanej linijki stwórz obiekt klasy `Product`.
- Przy odczytywaniu danych wykorzystaj:
  - Metodę `String.split()`. Służy ona do podziału łańcucha na kawałki. Przyjmuje jako argument wyrażenie regularne, zwraca tablicę stringów:

```
String linia = "10;poniedziałek";
String[] pola = s.split(";");
System.out.println(pola[0]);   // 10
System.out.println(pola[1]);   // poniedziałek
```

  - Metodę `Double.parseDouble()` do zamiany tekstu na liczbę:

```
String s = "127.50";
double d = Double.parseDouble(s);
```

- Odczytane produkty przechowaj w liście.
- Stwórz dodatkowo mapy:
  - mapa, która pozwoli na szybkie znalezienie produktu po id (id produktu zmapowane na obiekt `Product`- `Map<int,Product>`);
  - mapa, która pozwoli na szybkie pobranie listy produktów w kategorii (nazwa kategorii zmapowana na listę produktów

`Map<String,List<Product>>`). Najlepiej napisz to w taki sposób, aby dodanie nowej kategorii w pliku nie wymagało ponownej kompilacji programu.