Notatki - Comparator i Comparable

Spis treści

W jaki sposób sortować obiekty?

W przypadku liczb, sortowanie jest dosyć naturalne, można rosnąco albo malejąco. W przypadku Stringów, raczej też, możemy najpierw wielkie litery, potem małe litery zgodnie z kolejnością alfabetyczną, a potem np. cyfry. Ale co jeżeli chcemy posortować Koty, Psy, Samochody albo Krzesła?

W tym samym warsztacie poznamy też pewne rodzaje kolekcji, które automatycznie sortują dodane elementy, czyli muszą już wiedzieć w jakiś sposób jak mają posortować kolekcję, jeżeli dodamy do niej np. 4 monitory.

Wtedy z pomocą przychodzą nam 2 mechanizmy, `Comparator` oraz `Comparable`.

Na potrzeby tego wpisu wprowadzę klasę Cat, która będzie w tej formie używana później.

```
class Cat {

    private Integer weight;

    public Cat(final Integer weight) {
        this.weight = weight;
    }

    public Integer getWeight() {
        return weight;
    }

    @Override
    public String toString() {
        return "Cat{" +
            "weight=" + weight +
            '}';
    }
}
```

Comparator

Załóżmy, że mamy Listę kotów `List<Cat>` i chcemy ją w jakiś sposób posortować. Spójrzmy na ten przykład:

```
public class Example {

    public static void main(String[] args) {
        Comparator<Cat> comparator = new Comparator<Cat>() {
            @Override
            public int compare(final Cat o1, final Cat o2) {
                return o1.getWeight() - o2.getWeight();
            }
        };

        List<Cat> cats = new ArrayList<>();
        cats.add(new Cat(2));
        cats.add(new Cat(1));
        cats.add(new Cat(4));
        cats.add(new Cat(3));

        System.out.println("Before: " + cats);
        Collections.sort(cats, comparator);
        System.out.println("After: " + cats);
    }
}
```

Przykład powyżej wykorzystuje mechanizm klasy anonimowej. `Comparator` jest interfejsem funkcyjnym, oznacza to, że można go zaimplementować za pomocą lambdy. (Dokładnie zostanie to wyjaśnione jak dojdziemy do programowania funkcyjnego). Następne przykłady będą zatem implementowane wykorzystując lambdę.

```
public class Example {

    public static void main(String[] args) {
        Comparator<Cat> comparator1 = (o1, o2) -> o1.getWeight() - o2.getWeight();

        List<Cat> cats = new ArrayList<>();
        cats.add(new Cat(2));
        cats.add(new Cat(1));
        cats.add(new Cat(4));
        cats.add(new Cat(3));

        System.out.println("Before: " + cats);
        Collections.sort(cats, comparator1);
        System.out.println("After: " + cats);
    }
}
```

Sortujemy tutaj koty na podstawie ich wagi rosnąco.

Zgodnie z dokumentacją metody `int compare(T o1, T o2)` - metoda ta zwraca integer, który może być odpowiednio ujemny, zerowy lub dodatni. Jeżeli wynik jest:

- ujemny to pierwszy argument jest mniejszy niż drugi,
- zerowy to pierwszy argument jest równy drugiemu,
- dodatni to pierwszy argument jest większy niż drugi.

Biorąc to pod uwagę, jeżeli chcemy posortować elementy rosnąco, metoda `int compare(T o1, T o2)` musi być napisana tak:

```
Comparator<Cat> comparator1 = (o1, o2) -> o1.getWeight() - o2.getWeight();
```

Jeżeli natomiast chcielibyśmy posortować elementy malejąco, metoda `int compare(T o1, T o2)` musi być napisana tak:

```
Comparator<Cat> comparator1 = (o1, o2) -> o2.getWeight() - o1.getWeight();
```

Comparable

Comparable jest natomiast interfejsem, który z założenia ma być implementowany przez klasę, której obiekty chcemy sortować, czyli:

```
class Cat implements Comparable<Cat> {

    private Integer weight;

    public Cat(final Integer weight) {
        this.weight = weight;
    }

    public Integer getWeight() {
        return weight;
    }

    @Override
    public String toString() {
        return "Cat{" +
            "weight=" + weight +
            '}';
    }

    @Override
    public int compareTo(final Cat o) {
        return this.getWeight() - o.getWeight();
    }
}
```

Implementując interface `Comparable` musimy zaimplementować metodę `compareTo()`, co do której dokumentacja mówi, że metoda ta zwraca integer, który może być odpowiednio ujemny, zerowy lub dodatni. Jeżeli wynik jest:

- ujemny to pierwszy argument jest mniejszy niż drugi,

- zerowy to pierwszy argument jest równy drugiemu,
- dodatni to pierwszy argument jest większy niż drugi.

Przy czym w tym przypadku porównujemy obiekt, na którym operujemy, poprzez `this`, z obiektem, który przychodzi do nas jako argument metody `compareTo()`.

Biorąc to pod uwagę, jeżeli chcemy posortować elementy rosnąco, metoda `int compareTo(O o)` musi być napisana tak:

```
@Override
public int compareTo(final Cat o) {
    return this.getWeight() - o.getWeight();
}
```

Jeżeli natomiast chcielibyśmy posortować elementy malejąco, metoda `int compareTo(O o)` musi być napisana tak:

```
@Override
public int compareTo(final Cat o) {
    return o.getWeight() - this.getWeight();
}
```

W praktyce, możemy wtedy wywołać `Collections.sort(cats);` bez podawania `Comparatora` jako argument wywołania. Tutaj trzeba uważać, bo wywołanie `Collections.sort(cats);` na kolekcji obiektów, które nie implementują interfejsu `Comparable` spowoduje błąd kompilacji.

```
public class Example {

    public static void main(String[] args) {
        List<Cat> cats = new ArrayList<>();
        cats.add(new Cat(2));
        cats.add(new Cat(1));
        cats.add(new Cat(4));
        cats.add(new Cat(3));

        System.out.println("Before: " + cats);
        Collections.sort(cats);
        System.out.println("After: " + cats);
    }
}
```

No to czym one się różnią?

Jeżeli implementujemy `Comparable`, to mamy tylko jeden sposób sortowania obiektów, bo zawsze mamy zaimplementowaną metodą `compareTo(O o)` w jeden i ten sam sposób. `Comparator` natomiast pozwala nam posortować tę samą kolekcję na kilka różnych sposobów.

Wiedząc już czym jest `Comparator` i czym jest `Comparable` możemy pokusić się o ich porównanie:

Tabela 1. Porównanie Comparator vs Comparable

Comparator	Comparable
`Comparator` znajdziemy w paczce `java.util`	`Comparable` znajdziemy w paczce `java.lang`
Stosując `Comparator` wykorzystamy metodę `Collections.sort(list, comparator)`	Stosując `Comparable` wykorzystamy metodę `Collections.sort(list)`
Stosując `Comparator` implementujemy metodę `compare()`	Stosując `Comparable` implementujemy metodę `compareTo()`
Stosując `Comparator` **nie** modyfikujemy kodu klasy, czyli **nie** wpływamy na kod źródłowy klasy	Stosując `Comparable` modyfikujemy kod klasy, czyli wpływamy na kod źródłowy klasy
Stosując `Comparator` możemy z powodzeniem sortować obiekty klas, których nie napisaliśmy samodzielnie, czyli klasy z bibliotek zewnętrznych (o takich rzeczach będzie potem). Wynika to z tego, że nie możemy modyfikować kodu źródłowego klas z bibliotek zewnętrznych, a `Comparator` tego nie robi	Stosując `Comparable` problematyczne okazuje się sortowanie obiektów klas, których nie napisaliśmy samodzielnie, czyli klas z bibliotek zewnętrznych (o takich rzeczach będzie potem). Wynika to z tego, że nie możemy modyfikować kodu źródłowego klas z bibliotek zewnętrznych, a `Comparable` potrzebuje modyfikować kod klasy do poprawnego działania
Stosując `Comparator` możemy w prosty sposób określić sekwencję pól, na podstawie których sortujemy, czyli np. `name`, `surname`, `age` itp	Stosując `Comparable` również możemy to zrobić, natomiast jest to bardziej skomplikowane
Wykorzystując `Comparator` w jego implementacji nie będziemy stosować `Comparable`	Wykorzystując `Comparable` w jego implementacji możemy zastosować `Comparator`

Natural Ordering

Stosując `Comparator` natkniesz się na taki zapis:

```
Comparator.naturalOrder();
```

Wyjaśnijmy sobie zatem czym jest **natural order**, zanim natomiast to zrobimy, dodajmy jeszcze pojęcie **lexicographical order**.

Lexicographical order

Lexicographical order (porządek leksykograficzny) jest porządkiem alfabetycznym. Żeby lepiej to zrozumieć, przytoczę fragment dokumentacji metody `compareTo()` w klasie `String`.

> Compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings. The character sequence represented by this String object is compared lexicographically to the character sequence represented by the argument string.

Czyli tego rodzaju porównanie będzie się opierało na tablicy Unicode. Wartości w tej tablicy będą posortowane zgodnie z kolejnością: cyfry, duże litery, małe litery, duże polskie znaki, małe polskie znaki. Żeby to zobrazować, uruchom poniższy przykład.

```
List<String> list = new ArrayList<>();
list.add("text11");
list.add("text2");
list.add("aa");
list.add("b");
list.add("B");
list.add("Ó");
list.add("ó");
list.add("Ł");
list.add("ł");

System.out.println("Before: " + list);
Collections.sort(list);
System.out.println("After: " + list);
```

Na ekranie zostanie wydrukowane:

```
Before: [text11, text2, aa, b, B, Ó, ó, Ł, ł]
After: [B, aa, b, text11, text2, Ó, ó, Ł, ł]
```

Zgodnie z tym co jesteśmy nauczeni, `2` powinno być przed `11`, więc `text2` powinno być przed `text11`. Zgodnie z kolejnością alfabetyczną `ł` powinno być przed `t`, a tutaj widać, że jest za nim. Wszystkie te "dziwne" zachowania wynikają z tego, że domyślne sortowanie `Stringów` w Java odbywa się na podstawie tablicy `Unicode`. Wrócimy do tematu czym jest tablica `Unicode` w warsztacie o operacjach na plikach na dysku komputera. Zgodnie z tą tablicą, przykład `11` i `2` należy traktować analogicznie jak `b` i `aa`, gdzie `2` jest jak `b`, a `11` jest jak `aa`.

Natural order

Zostało wprowadzone również takie pojęcie jak **natural order** Wikipedia. Taka kolejność sortowania powinna być bardziej zrozumiała dla człowieka, czyli z tym podejściem do sortowania powinniśmy zobaczyć kolejność `text2`, a później `text11`.

Java jest tutaj myląca. Jeżeli w przykładzie powyżej, użyjemy do sortowania `Comparator.naturalOrder()`, czyli przepiszemy sortowanie w ten sposób:

```
Collections.sort(list, Comparator.naturalOrder());
```

to na ekranie zostanie wydrukowane:

```
Before: [text11, text2, aa, b, B, Ó, ó, Ł, ł]
After: [B, aa, b, text11, text2, Ó, ó, Ł, ł]
```

Czyli to samo prawda? A przecież **natural order** miał załatwić sprawę. I tutaj właśnie jest ta rozbieżność. Pojęcia **natural order** oznacza sortowanie bardziej naturalne dla człowieka. Natomiast metoda `Comparator.naturalOrder()` pod spodem woła metodę `compareTo()` z klasy, której obiekty mają być sortowane. Jak to działa dla klasy `String` zostało wspomniane wcześniej. Stąd otrzymujemy taki wynik.

Jeżeli ktoś ma ochotę dalej pogrzebać to tutaj umieszczam link do **Stackoverflow**.

Gdybyśmy chcieli porównać ze sobą sortowanie **alfabetyczne** z **natural order** - wyglądałoby to w ten sposób: