Comparable i Comparator

Czego się dowiesz

- Jakie możliwości dają interfejsy Comparable i Comparator,
- w jaki sposób sortować tablice obiektów (nie typów prostych ani obiektów typu String).

Wstęp

W lekcji dotyczącej operacji na tablicach pokazałem ci, że możliwe jest posortowanie tablicy przy pomocy metody Arrays.sort(). O ile jednak w przypadku tablic typów prostych, czy obiektów typu String nie ma z tym problemu, tak przy sortowaniu innych obiektów (np. Person, Car, etc) pojawia się spory problem, ponieważ nie jest w żaden sposób określony naturalny porządek elementów (np. alfabetyczny) i nie wiadomo po którym polu takiego obiektu chcielibyśmy sortować.

W Javie w celu rozwiązania tego problemu można skorzystać z dwóch interfejsów: Comparable lub Comparator.

Interfejs Comparable

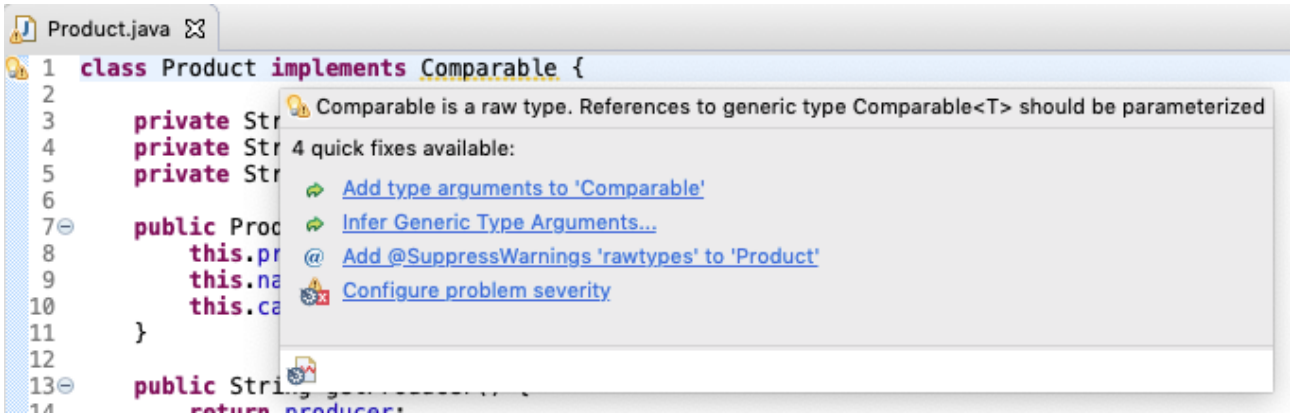
Jeżeli zależy nam jedynie na tym, aby był ustalony jeden, naturalny porządek elementów i nie potrzebujemy sortowania według kilku różnych kryteriów, np. rosnąco, malejąco, po imieniu lub po nazwisku itp., to najlepszym rozwiązaniem będzie zaimplementować przez klasę interfejsu Comparable.

Zdefiniuj klasę Product, która posiada atrybuty takie jak producer (producent), name (nazwa produktu) i category (kategoria produktu). Wygeneruj metodę toString(), która wyświetli dane w kolejności kategoria, producent, nazwa i dodaj do sygnatury klasy informację o implementowaniu interfejsu Comparable (implements Comparable).

Product.java

```
class Product implements Comparable {

    private String producer;
    private String name;
    private String category;

    public Product(String producer, String name, String category) {
        this.producer = producer;
        this.name = name;
        this.category = category;
    }

    public String getProducer() {
        return producer;
    }

    public void setProducer(String producer) {
        this.producer = producer;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getCategory() {
        return category;
    }

    public void setCategory(String category) {
        this.category = category;
    }

    @Override
    public String toString() {
        return "Product [category=" + category + ", producer=" +
producer + ", name=" + name + "]";
    }

    @Override
    public int compareTo(Object o) {
        // TODO Auto-generated method stub
        return 0;
    }

}
```

W interfejsie tym zdefiniowana jest tylko jedna metoda **compareTo()**, co przetłumaczyć można na "porównaj z". Ponieważ nie jest to metoda domyślna, musimy ją zaimplementować w naszej klasie. Jeśli korzystamy z eclipse, to w nagłówku klasy zobaczymy ostrzeżenie:



Problem polega na tym, że interfejs Comparable jest typem generycznym i w takiej sytuacji powinniśmy określić z obiektami jakiego typu będziemy chcieli porównywać obiekty typu Product. W naszym przypadku oczywiście będziemy porównywali obiekty Product z innymi obiektami Product. Dzięki określeniu parametru generycznego metoda compareTo() będzie miała parametr typu Product, a nie Object, co zaoszczędzi nam niepotrzebnego rzutowania.

Poprawiona klasa Product wygląda więc następująco:

Product.java

```
class Product implements Comparable<Product> {

    private String producer;
    private String name;
    private String category;

    public Product(String producer, String name, String category) {
        this.producer = producer;
        this.name = name;
        this.category = category;
    }

    public String getProducer() {
        return producer;
```

```
    }

    public void setProducer(String producer) {
        this.producer = producer;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getCategory() {
        return category;
    }

    public void setCategory(String category) {
        this.category = category;
    }

    @Override
    public String toString() {
        return "Product [category=" + category + ", producer=" +
producer + ", name=" + name + "]";
    }

    @Override
    public int compareTo(Product o) {
        // TODO Auto-generated method stub
        return 0;
    }

}
```

W metodzie compareTo(arg) musimy teraz zdefiniować zasady sortowania i zwrócić jedną z wartości:

- liczbę ujemną, jeśli obiekt przekazany jako argument (arg) jest "mniejszy", ma być poprzednikiem obiektu, z którym go porównujemy (this),
- liczbę 0 jeśli porównywane obiekty są sobie równe,
- liczbę dodatnią, jeśli obiekt oryginalny (this) jest "większy", ma być następnikiem obiektu przekazanego jako argument (arg).

W przypadku sortowania w klasie takiej jak Product możemy zastosować następującą zasadę:

- sortujemy po nazwie kategorii produktu,
- jeśli kilka produktów jest w tej samej kategorii, sortujemy po nazwie producenta,
- jeśli nazwa producenta kilku produktów jest taka sama, to sortujemy po nazwie towaru.

Zaktualizowana metoda compareTo() będzie wyglądała następująco:

```
@Override
public int compareTo(Product p) {
    int categoryCompare = category.compareTo(p.getCategory());
    if (categoryCompare != 0) {
        return categoryCompare;
    }
    int producerCompare = producer.compareTo(p.getProducer());
    if (producerCompare != 0) {
        return producerCompare;
    }
    return name.compareTo(p.getName());
}
```

Porównujemy po kolei każde kolejne pole. Ponieważ wszystkie są typu String, to nie możemy ich porównać po prostu operatorem większe mniejsze. Musimy wykorzystać metodę compareTo() zdefiniowaną w klasie String, która porównuje napisy na podstawie ich kodów kolejnych znaków. Jest to porównywanie prawie w porządku alfabetycznym o ile nie używamy polskich znaków diakrytycznych. Przykładowo:

```
int compareTo1 = "abc".compareTo("xyz"); // -23
int compareTo2 = "xyz".compareTo("abc"); // 23
int compareTo3 = "xyz".compareTo("xyz"); // 0
```

Jeżeli porównanie kategorii zwróci 0 to przechodzimy do porównywania producentów. Jeżeli wynik jest różny od 0, to zwracamy wynik tego porównania, a jeśli nie to porównujemy jeszcze nazwy produktów.

Jeżeli stworzymy teraz tablicę kilku elementów typu Product i posortujemy ją za pomocą Arrays.sort(), to otrzymamy obiekty posortowane po kategorii, producencie, a na końcu nazwie.

ProductCatalog.java

```
import java.util.Arrays;

class ProductCatalog {
    public static void main(String[] args) {
        Product[] products = new Product[7];
        products[0] = new Product("Amino", "Zupa pomidorowa", "Zupy");
        products[1] = new Product("Amino", "Zupa ogórkowa", "Zupy");
        products[2] = new Product("WINIARY", "Zupa pomidorowa", "Zupy");
        products[3] = new Product("WINIARY", "Zupa pomidorowa", "Zupy błyskawiczne");
        products[4] = new Product("WINIARY", "Rosół", "Zupy");
        products[5] = new Product("Knorr", "Placki ziemniaczane", "Dania obiadowe");
        products[6] = new Product("Knorr", "Racuchy", "Dania obiadowe");

        System.out.println("Nieposortowana: ");
        for (Product p : products) {
            System.out.println(p);
        }

        Arrays.sort(products);
        System.out.println("Posortowana: ");
        for (Product p : products) {
            System.out.println(p);
        }
    }
}
```

W wyniku działania powyższego kodu możemy zobaczyć następujący wydruk:

co potwierdza, że tablica faktycznie została posortowana zgodnie z zadanymi kryteriami. Jeżeli w klasie mamy pola typów prostych to sytuacja jest dużo prostsza, bo do ich porównania można wykorzystać po prostu operatory większy/mniejszy, nie trzeba wywoływać żadnej metody compareTo().

Interfejs Comparator

Problemem powyższego rozwiązania jest fakt, że korzystając z interfejsu Comparable możemy zdefiniować tylko jeden sposób sortowania, który będziemy rozumieli jako naturalny porządek dla danej klasy. Co jednak w sytuacji, gdy chcielibyśmy posortować jakąś tablicę po nazwie produktu, a innym razem po nazwie producenta? Przedefiniowanie metody compareTo() i ponowna kompilacja programu za każdym razem raczej nie wchodzi w grę.

Jeśli jednak spojrzymy na inną wersję metody Arrays.sort(), to zauważymy, że dostępna jest także taka, która przyjmujące tablicę oraz drugi argument w postaci obiektu klasy implementującej interfejs **Comparator**(tzw. komparatora)

Dzięki temu możemy zdefiniować kilka klas, które pozwolą nam określić różny porządek obiektów, które sortujemy. Jest to miejsce, w którym wygodne jest zastosowanie statycznej klasy zagnieżdżonej lub klasy anonimowej (a od Javy 8 także wyrażenia lambda) w przypadku prostego porównania.

Przykład 1 - zagnieżdżona klasa wewnętrzna

Jako pierwszy przykład pokażemy jak zastosować Comparator jako statyczną klasę zagnieżdżoną. Dlaczego statyczną? Ponieważ dzięki temu unikamy konieczności tworzenia obiektu klasy nadrzędnej. Większość klasy Product zostaje bez zmian, poniżej pokazano dodatkowy kod, który pozwoli nam posortować produkty po nazwie produktu.

Product.java

```
import java.util.Comparator;

class Product implements Comparable<Product> {

    // reszta bez zmian

    public static class ProductNameComparator implements Comparator<Product> {
        @Override
        public int compare(Product p1, Product p2) {
            return p1.getName().compareTo(p2.getName());
        }
    }

}
```

W interfejsie Comparator zdefiniowana jest jedna metoda abstrakcyjna **compare()**, którą musimy zaimplementować. Ponieważ interfejs ten również jest generyczny to pozwala nam na zdefiniowanie typu, jaki chcemy porównywać. Zwróć uwagę, że nazwa metody to samo compare(), a nie compare**To**().

Wykorzystanie takiego komparatora w metodzie Arrays.sort() wyglądać będzie następująco:

```
Arrays.sort(products, new Product.ProductNameComparator());
```

Przykład 2 - klasa anonimowa

W przypadku tak prostego komparatora jak powyżej warto także rozważyć zastosowanie anonimowej klasy wewnętrznej, która implementuje interfejs Comparator. Będzie to rozwiązanie sensowne w przypadku, gdy sortowanie odbywa się tylko w jednym miejscu programu i wiemy, że raczej nam się już nigdy nie przyda.

Prosta klasa anonimowa sortująca produkty zgodnie z nazwą producenta wyglądałaby następująco:

```
Arrays.sort(products, new Comparator<Product>() {
    @Override
    public int compare(Product o1, Product o2) {
        return o1.getProducer().compareTo(o2.getProducer());
    }
});
```

Dzięki takiemu rozwiązaniu komparator tworzony jest tylko na potrzeby tego pojedynczego sortowania. Z powodu budowy klasy anonimowej raczej nie jest wskazane wykorzystywanie tego rozwiązania przy bardziej rozbudowanych porównaniach.

Przykład 3 - osobna klasa

Oczywiście komparator może być także zdefiniowany jako klasyczna klasa w osobnym pliku.

ProductNameComparator.java

```
import java.util.Comparator;

class ProductNameComparator implements Comparator<Product> {
    @Override
    public int compare(Product p1, Product p2) {
        return p1.getName().compareTo(p2.getName());
    }
}
```

Jego użycie wygląda wtedy podobnie jak w przypadku klasy zagnieżdżonej:

```
Arrays.sort(products, new ProductNameComparator());
```

To, z którego podejścia zdecydujesz się skorzystać zależy tylko od ciebie i tego co uznasz za czytelniejsze w danym momencie.