Theory: Comparable

51 minutes

Skip this topic | Start practicing

While working with data, you will likely need to order it in a convenient way. For example, you may have to put numbers in ascending order, group lines in alphabetical order, and organize anything you work with by date, by price, or by other custom characteristics.

In Java, it's possible to implement various sorting algorithms for any type of data. What if you have to work with custom types, sort elements of a collection, and try to compare objects that are not directly comparable? That's where the `Comparable` interface comes in handy. In this topic, we will learn all about this interface and its `compareTo()` method.

§1. Preparing to compare

Let's look at an example. We created a list of `Integer`'s, added some elements and then sorted them.

```
public static void main(String[] args) {
    List<Integer> list = new ArrayList<>();
    list.add(55);
    list.add(13);
    list.add(47);

    Collections.sort(list);
    System.out.println(list);
}
```

As expected, we get:

```
[13, 47, 55]
```

Now, let's create a simple class `Car` where we want to sort cars by their numbers.

```
public class Car {
    private int number;
    private String model;
    private String color;
    private int weight;

    // constructor

    // getters, setters
}
```

Now we try to write some code for the `main` method, create our collection and sort it using the `Collections.sort()` method.

```
public static void main(String[] args) {
    List<Car> cars = new ArrayList<>();
    Car car1 = new Car(876, "BMW", "white", 1400);
    Car car2 = new Car(345, "Mercedes", "black", 2000);
    Car car3 = new Car(470, "Volvo", "blue", 1800);
    cars.add(car1);
    cars.add(car2);
    cars.add(car3);

    Collections.sort(cars);
    System.out.println(cars);
}
```

3 required topics

- ✓ Interface — In project
- ✓ Boxing and unboxing — In project — 2
- ✗ The List interface

4 dependent topics

- Memento
- ✓ Processing strings
- BigInteger
- Comparator

Table of contents:

- ↑ Comparable
- §1. Preparing to compare
- §2. Comparable interface
- §3. Implementing the compareTo method
- §4. Conclusion
- Discussion

As a result, we get a compilation error:

```
The method sort(List<T>) in the type Collections
  is not applicable for the arguments (ArrayList<Car>)
```

The reason for this is that standard classes like `Integer`, `String` and so on implement a special interface, so we can compare them without any problems. As for our custom class `Car`, it doesn't work like that. Let's see how we can fix this.

§2. Comparable interface

`Comparable` provides the `compareTo()` method which allows comparing an object with other objects of the same type. It's also important to comply with the conditions: all objects can be compared to other objects of the same type in the most widely used way, which means `compareTo()` should be consistent with the `equals` method. A sequence of data has the **natural ordering**, if for each 2 elements `a` and `b`, where `a` is located to the left of `b`, the condition is true: `a.compareTo(b) <= 0`

It's easy to understand how to compare an `Integer` or `String` because they already implement the `Comparable` interface, but how do we compare objects of our custom type? We can do it in different ways depending on the task. We can compare them by any single field or several fields.

To be able to sort, we must rewrite our `Car` class using the `Comparable` interface. For example, we can compare our `Car` objects by their `number`. Here's how you can implement it:

```
1  public class Car implements Comparable<Car> {
2
3      private int number;
4      private String model;
5      private String color;
6      private int weight;
7
8      // constructor
9
10     // getters, setters
11
12     @Override
13     public int compareTo(Car otherCar) {
14
       return Integer.valueOf(getNumber()).compareTo(otherCar.getNumber());
15     }
16
17 }
```

Now if we run our new code we get the correct result.

§3. Implementing the compareTo method

Let's talk about the `compareTo()` method. It compares the current object with the object sent as a parameter. To implement it correctly we need to make sure that the method returns:

- A positive integer (for example, 1), if the current object is greater;
- A negative integer (for example, -1), if the current object is less;
- Zero, if they are equal.

Below you can see an example of how the `compareTo()` method is implemented in the `Integer` class.

```
@Override
public int compareTo(Integer anotherInteger) {
    return compare(this.value, anotherInteger.value);
}

public static int compare (int x, int y) {
    return (x < y) ? -1 : ((x == y) ? 0 : 1);
}
```

There are some other rules for implementing the `compareTo()` method. To demonstrate them, imagine we have a class called `Coin:`

```
class Coin implements Comparable<Coin> {
    private final int nominalValue;    // nominal value

private final int mintYear;        // the year the coin was minted

    Coin(int nominalValue, int mintYear) {
        this.nominalValue = nominalValue;
        this.mintYear = mintYear;
    }

    @Override
    public int compareTo(Coin other) {
        // This method we have to implement
    }


// We consider two coins equal if they have the same nominal value
    @Override
    public boolean equals(Object that) {
        if (this == that) return true;

  if (that == null || getClass() != that.getClass()) return false;
        Coin coin = (Coin) that;
        return nominalValue == coin.nominalValue;
    }

    // getters, setters, hashCode and toString
}
```

Let's create some objects of `Coin`.

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

    Coin big = new Coin(25, 2006);
    Coin medium1 = new Coin(10, 2016);
    Coin medium2 = new Coin(10, 2001);
    Coin small = new Coin(2, 2000);
}
```

One of the rules is to keep the `compareTo()` implementation consistent with the implementation of the `equals()` method. For example:

- `medium1.compareTo(medium2) == 0` should have the same boolean value as `medium2.equals(medium1)`

If we compare our coins and `big` is bigger than `medium1` and `medium2` is bigger than `small`, then `big` is bigger than `small`:

- `(big.compareTo(medium1) > 0 && medium1.compareTo(small) > 0)` implies `big.compareTo(small) > 0`

`medium1` is bigger than `small`, hence `small` is smaller than `big`:

- `big.compareTo(small) > 0` and `small.compareTo(big) < 0`

if `medium1` is equal to `medium2`, they both must be bigger or smaller than `small` and `big` respectively:

- `medium1.compareTo(medium2) == 0` implies that `big.compareTo(medium1) > 0` and `big.compareTo(medium2) > 0` or `small.compareTo(medium1) < 0` and `small.compareTo(medium2) < 0`

This will ensure that we can safely use such objects in sorted sets and sorted maps.

In this case, we can comply with all these requirements if we compare our coins by their nominal value:

```
class Coin implements Comparable<Coin> {

    // fields, constructor, equals, hashCode, getters and setters

    @Override
    public int compareTo(Coin other) {
        if (nominalValue == other.nominalValue) {
            return 0;
        } else if (nominalValue < other.nominalValue) {
            return -1;
        } else {
            return 1;
        }
    }

    @Override
    public String toString() {

return "Coin{nominal=" + nominalValue + ", year=" + mintYear + "}";
    }
}
```

Now we can add the coins to a list and sort them:

```
List<Coin> coins = new ArrayList<>();

coins.add(big);
coins.add(medium1);
coins.add(medium2);
coins.add(small);

Collections.sort(coins);
coins.forEach(System.out::println);
```

In the output, we can see that the coins have been successfully sorted:

```
Coin{nominal=2, year=2000}
Coin{nominal=10, year=2016}
Coin{nominal=10, year=2001}
Coin{nominal=25, year=2006}
```

§4. Conclusion

In this topic, we explored how to use the `Comparable` interface in our custom classes to define natural ordering algorithms. We also learned how to properly implement the `compareTo` method.

Comments (6) Hints (0) Useful links (2) Show discussion