JAVA SE

Quiz yourself: Sorting lists by multiple criteria

Achieving the desired results by using the Comparator factory mechanisms

by Mikalai Zaikin and Simon Roberts

August 3, 2021

If you have worked on our quiz questions in the past, you know none of them is easy. They model the difficult questions from certification examinations. We write questions for the certification exams, and we intend that the same rules apply: Take words at their face value and trust that the questions are not intended to deceive you but to straightforwardly test your knowledge of the ins and outs of the language.

Given this `Animal` class

```
public class Animal {
  private String name;
  private int numOfLegs;
  private int numOfWings;
  public Animal(String name, int numOfLegs, int nu
    this.name = name;
    this.numOfLegs = numOfLegs;
    this.numOfWings = numOfWings;
  }
  … // getters and setters
  @Override
  public String toString() {
    return name;
  }
}
```

and this code fragment

```
List<Animal> listOfAnimal = List.of(
  new Animal("Spider", 8, 0),
  new Animal("Fly", 6, 2),
  new Animal("Dragonfly", 6, 4),
  new Animal("Worm", 0, 0)
);
… // add sorting code here
```

Which code fragment allows you to sort animals in the following order: Worm, Fly, Dragonfly, Spider? Choose one.

A.

```
listOfAnimal.stream().sorted(Comparator.comparing(a
-> a.getNumOfLegs()).thenComparing(a ->
a.getNumOfWings())).collect(Collectors.toList());
```

The answer is A.

B.

```
listOfAnimal.stream().sorted(Comparator.comparing(a
-> a.getNumOfWings()).thenComparing(a ->
a.getNumOfLegs())).collect(Collectors.toList());
```

The answer is B.

C.

```
listOfAnimal.stream().sorted(Comparator.comparing(a
-> a.getNumOfLegs())).sorted(Comparator.comparing(a
-> a.getNumOfWings())).collect(Collectors.toList());
```

The answer is C.

D.

```
listOfAnimal.sort((la, ra) -> {
  if (la.getNumOfLegs() == ra.getNumOfLegs()) {
    return Integer.compare(la.getNumOfWings(),
                           ra.getNumOfWings());
  } else {
    return Integer.compare(la.getNumOfLegs(),
                           ra.getNumOfLegs());
  }
});
```

The answer is D.

E. None of the above

The answer is E.

Answer. The task is to sort animals by multiple criteria. This is similar to sorting a list of people by last names, and subsorting by first names if the last names are the same. In this case, you must sort by number of legs and, if the number of legs is the same, perform a subsort by number of wings. This will produce the required ordering.

Option A looks tempting; it uses the factory mechanisms in the `Comparator` in a way that suggests that it would sort first by number of legs and then subsort by number of wings. However, when you chain these methods, the type inferencing mechanism no longer deduces that the comparator will get an `Animal`. Instead, it decides the items might be `Object`. Of course, if the lambda parameter `a` is of type `Object`, it

does not have `getNumOfLegs` or `getNumOfWings`. Because of this, compilation fails and option A is incorrect.

By the way, you could fix this code by any means that avoids using type inferencing on the entire chain. One simple solution—there are at least two other good contenders, but we'll leave them as exercises for the reader—is to explicitly specify that the lambda parameters' types are `Animal`. The following code would compile successfully, and it would produce the desired outcome:

```
listOfAnimal.stream().sorted(Comparator.comparing(
```

Option B fails on two counts. Most obviously, the order is incorrect: It sorts by number of wings first and then by number of legs. Such sorting would put the items in the order `[Worm, Spider, Fly, Dragonfly]`, which is incorrect. Of course, the type inference problem also applies here, and this code does not compile either, so option B is definitely incorrect.

Option C performs two subsequent unrelated sorts. The `Stream.sorted` method states that for ordered streams such as this, which is drawn from a `List`, the sort is *stable*. This means that if a second sort does not need to change the relative position of two items, it will not. The effect is that the first sort becomes the minor sort, and the second sort is the major sort. If you prefer, you can think of this as being that the first sort "shows through" when, but only when, the second sort does not have to change things. Therefore, this produces the same order as option B: `[Worm, Spider, Fly, Dragonfly]`. From this you can see that Option C is also incorrect.

Remember that sort operations are computationally expensive, so whenever possible, avoid doing things twice.

Option D fails because `List.of` creates an *unmodifiable* list, but the `List.sort()` method attempts to perform the sorting in place by modifying the original list. Consequently, this code will fail with `java.lang.UnsupportedOperationException`. From this you know that option D is incorrect.

There are two side notes here.

First, if the list were made modifiable, as follows, option D would produce the correct result:

```
listOfAnimal = new ArrayList<>(listOfAnimal);
```

Second, the notion of an *unmodifiable* list might seem a bit odd. Why not simply call it an *immutable* list? This is because the list itself cannot control modifications made to any mutable items that it contains. So, the `List` API designers decided to use the term *unmodifiable* to highlight the difference, notably that it's not necessarily safe to share these objects without some thought. There are more details in the API documentation.

Based on the above you can see that all four options A through D fail one way or another. Therefore, option E is correct.

Conclusion. The correct answer is option E.

Related quizzes

- Quiz yourself: Manipulating Java lists—and views of lists
- Quiz yourself: Declare and use List and ArrayList instances
- Quiz yourself: Using subclasses and covariant return types

Mikalai Zaikin

Mikalai Zaikin is a lead Java developer at IBA IT Park in Minsk, Belarus. During his career, he has helped Oracle with development of Java certification exams, and he has been a technical reviewer of several Java certification books, including three editions of the famous *Sun Certified Programmer for Java* study guides by Kathy Sierra and Bert Bates.

Simon Roberts

Simon Roberts joined Sun Microsystems in time to teach Sun's first Java classes in the UK. He created the Sun Certified Java Programmer and Sun Certified Java Developer exams. He wrote several Java certification guides and is currently a freelance educator who publishes recorded and live video training through Pearson InformIT (available direct and through the O'Reilly Safari Books Online service). He remains involved with Oracle's Java certification projects.

Share this Page