QUIZ

Quiz Yourself: Using Collectors (Advanced)

Where care is needed to get the results you expect from the Collectors class

by Simon Roberts and Mikalai Zaikin

August 26, 2019

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. The levels marked “intermediate” and “advanced” refer to the exams, rather than the questions. Although in almost all cases, “advanced” questions will be harder. 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 straightforwardly test your knowledge of the ins and outs of the language.

The objective is to save results to a collection using the `collect` method and group or partition data using the `Collectors` class. Given the following `Student` class and given a `Stream<Student> s` that is initialized, unused, in scope, and contains students of varying ages:

```
class Student {
    private String name;
    private Integer age;

    Student(String name, Integer age) {
        this.name = name;
        this.age = age;
    }

    public String getName() { return name; }
    public Integer getAge() { return age; }
}
```

Which of the following code fragments prints the number of students under 18 and the number of students 18 years old or older in the best way? Choose one.

A.
```
s.collect(Collectors.groupingBy(
            Student::getAge() >= 18,
            Collectors.counting()))
 .forEach((c, d) -> System.out.println(d));
```

B.
```
s.collect(Collectors.groupingBy(
      a -> a.getAge() >= 18,
          Collectors.mapping(
              Student::getName,
              Collectors.counting())))
 .forEach((c, d) -> System.out.println(d));
```

C.

```
s.collect(Collectors.partitioningBy(
    a -> a.getAge() >= 18, Collectors.counting()))
 .forEach((c, d) -> System.out.println(d));
```

D.

```
List<Integer> l = Arrays.asList(0, 0);
s.forEach(w -> {
    if (w.getAge() >= 18) {
        l.set(1, l.get(1) + 1);
    } else {
        l.set(0, l.get(0) + 1);
    }});
l.forEach(System.out::println);
```

Answer. This question demonstrates several ways to collect data and group it according to a criterion. Options A, B, and C aim to do this using utilities from the `Collectors` class together with a `Stream.collect` method. Option D takes a handcrafted approach.

Let's address the easy option first. In option A, the code contains a significant syntax error. It attempts to use a method reference to invoke a method in this expression:

```
Student::getAge()
```

This syntax is illegal; method reference syntax cannot be used in this way, which tells you that option A is incorrect.

The exam generally tries to avoid "human compiler" type tests because such a skill isn't useful in these days of IDEs that check syntax as you type. If this were the only reason to reject this option, it would probably be rejected from the real exam. However, two additional reasons exist to reject this option. One is that the approach of another option is better designed; it's only a marginal improvement, but it's real, and you'll see it soon. The second reason is that there's another correct answer, and you must select only one. Therefore, finding the other answer should prompt you to pay close attention and notice the error. A hint here is that in this exam, it's unwise to simply select the first answer you see that appears to be correct. Unless you're short of time, take a moment to satisfy yourself that all the other answers are actually incorrect.

It's worth noting that if you replace the expression containing the syntax error

```
Student::getAge() >= 18
```

with this corrected version:

```
a -> a.getAge() >= 18
```

then option A would produce the correct output and only the question of design quality would remain.

For now, however, it's clear that option A should be rejected immediately and that you will find the better-designed approach and see why it's better designed.

The code in option B compiles successfully and prints the correct result. To understand this, let's examine the behaviors of some of the key features of the `Collections` class. The `groupingBy` method creates a `Collector` behavior that takes a function as argument. That function is called the *classifier*. For example, consider this code, which performs a `groupingBy` collection operation on a stream of names:

```
Stream.of("Fred", "Jim", "Sheila", "Chris",
          "Steve", "Hermann", "Andy", "Sophie")
      .collect(Collectors.groupingBy(n -> n.length(
```

The following expression is the classifier; it takes the name (as a `String`) and returns its length:

```
n -> n.length()
```

The effect would be to build a `Map<Integer, List<String>>` that looks like what's shown in **Table 1**:

KEY	VALUE
3	List["Jim"]
4	List["Fred", "Andy"]
5	List["Chris", "Steve"]
6	List["Sheila", "Sophie"]
7	List["Hermann"]

Table 1. Keys and their associated values

Notice how the `Map` contains each of the values returned by the classifier as a key, and the value associated with that key is a `List` containing all the stream items that produced that key.

A variant of the `groupingBy` behavior allows you to have a "downstream collector." Instead of simply adding the items that produce a particular key into a list of all such items, this allows you to use a subsequent collection operation to process the items. This is somewhat akin to a secondary stream process working on the items that would have been sent to the `List`. One collector that's commonly used in the downstream position is the `counting` collector. Using this in your process would change the values from being lists of names to being a count of the number of elements that would have been in the list.

This code

```
Stream.of("Fred", "Jim", "Sheila", "Chris",
          "Steve", "Hermann", "Andy", "Sophie")
      .collect(Collectors.groupingBy(
          n -> n.length(),
          Collectors.counting()))
```

would return the `Map<Integer, Long>` shown in **Table 2**:

KEY	VALUE
3	1
4	2
5	2
6	2
7	1

Table 2. Keys and their associated values when a downstream collector is used

Now, the behavior of option B is similar to this idea, but it has a couple of variations. First, the classifier—and, hence, the keys in the resulting `Map`—are boolean in nature, rather than numeric. That's fine, because it suits the purpose of classifying students who are over 18 and those who are not over 18. However, the code in option B actually has *two* downstream operations chained, rather than one. This is certainly permissible and can be very useful. But in this case, the first downstream collector actually performs a mapping operation that extracts from each `Student` object

the name of the `Student`. The second downstream collection counts the resulting names.

The mapping operation does not render the result incorrect; the code effectively counts the names of the students over or under 18 years of age and produces the same numbers. However, it's a waste of effort and, as such, the code is not optimally efficient, nor is it the most readable, because it includes distracting, irrelevant code that risks causing confusion. You will see shortly that another option avoids this wasted effort, and that option will show you that option B is incorrect. Despite producing the correct response, option B is not the most efficient option.

In addition to the `groupingBy` behavior factory in the `Collectors` class, there is another factory that produces a result that's somewhat similar. This factory is called `partitioningBy`, and the difference in its behavior is simply that instead of creating a `Map` with an arbitrary key type, it specifically creates a `Map` with a `Boolean` key. To support this, the classifier should be a predicate rather than a function. Option C uses the `partitioningBy` behavior and produces the correct output. It is better than option B for two reasons: First, it does not involve the wasted step of extracting names from the students. Second, `partitioningBy` is specifically provided exactly for the purpose of grouping by a simple true/false result and, as such, it is a (marginally) better design choice. By the same logic, it would be better than option A, even if that option had valid syntax.

Option D uses handcrafted code and might also produce the correct answer. The major problem is that this code operates by means of *side effects*. Specifically, it modifies variables outside a lambda expression. This kind of behavior makes code unsafe in concurrent or parallel execution; particularly in stream-based systems, this type of code should be avoided precisely because a stream can easily be run in a parallel mode.

It's worth noting that when the lambda expression (and before it, method local classes) was designed, this idea was loosely expressed by the prohibition on accessing other method local variables from inside lambdas or nested classes unless those variables are effectively final. The code in option D creates its side effects by using an effectively final reference to reach the mutable data in the `List` to which it refers.

The code compiles successfully, and if the stream ran sequentially, it would print the correct result. However, it is badly designed, and if the stream were parallel, it would fail. Because the design is unsound and would fail if the stream were parallel, option D is incorrect.

The correct answer is option C.

Also in This Issue

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.

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.

Share this Page