Imagine that the list produced by this code is always empty, even though you are using test data with sighting records that should definitely match. Clearly there must be something wrong with the **filter** operation, but you cannot immediately spot the problem. One of the easiest ways to get to the bottom of this would be to split the single filter operation into three separate filters, and to print out fields of the remaining elements after each filtration step. The following version does this:

```
List<Sighting> result =
  sightings.stream()
           .filter(record -> animal == record.getAnimal())
           .peek(r -> System.out.println(r.getAnimal())
           .filter(record -> area == record.getArea()
           .peek(r -> System.out.println(r.getArea())
           .filter(record -> spotter == record.getSpotter())
           .peek(r -> System.out.println(r.getDetails())
           .collect(Collectors.toList());
```

From this version you would be quite likely to find that the first filter operation is not passing on any records to the next, because the test for string equality has not been made using the **equals** method. Once that has been corrected, the tests could be rerun and eventually the **peek** operations removed.

A **peek** operation can also be a convenient way to create a position in the middle of a pipeline sequence for setting a breakpoint for a debugger. In this case, you would be more likely to be interested in the states of the objects rather than having something printed, so a consumer lambda that does nothing would be used as the parameter to **peek**, for instance:

```
peek(r -> { })
```

9.12 Choosing a debugging strategy

We have seen that several different debugging and testing strategies exist: written and verbal walkthroughs, use of print statements (either temporary or permanent, with enabling switches), interactive testing using the object bench, writing your own test class, and using a dedicated unit test class.

In practice, we would use different strategies at different times. Walkthroughs, print statements, and interactive testing are useful techniques for initial testing of newly written code, to investigate how a program segment works, or for debugging. Their advantage is that they are quick and easy to use, they work in any programming language, and they are (except for the interactive testing) independent of the environment. Their main disadvantage is that the activities are not easily repeatable. This is okay for debugging, but for testing we need something better: we need a mechanism that allows easy repetition for regression testing. Using unit test classes has the advantage that—once they have been set up—tests can be replayed any number of times.