Monitoring animal populations

Procedural version

As before, we will use an example program to introduce and discuss the new constructs. Here, we will look at a system to monitor animal populations.

An important element of animal conservation and the protection of endangered species is the ability to monitor population numbers, and to detect when levels are healthy or in decline. Our project processes reports of sightings of different types of animal, sent back by spotters from various different places. Each sighting report consists of the name of the animal that has been seen, how many have been seen, who is sending the report (an integer ID), which area the sighting was made in (an integer), and an indication of when the sighting was made (a simple numerical value which might be the number of days since the experiment started).

The simple data requirements make it easy for spotters in remote locations to send back relatively small amounts of valuable information — perhaps in the form of a text message — to a base that is then able to aggregate the data and create reports or direct the field workers to different areas.

This approach fits well with both highly managed projects — such as might be undertaken by a National Park and involve motion triggered cameras and people evaluating the cam- era footage — and with loosely organized crowd-sourcing activities—such as national bird-counting days, where a large group of volunteers uses phone apps to send data.

We provide a partial implementation of such a system under the package **`animalmonitoring.v1`**. Load the code into your project from *animal_monitoring.jar*.

The **`Sighting`** class records details of each sighting report, from a single spotter for a particular animal, once the sighting details have been processed. We will not be concerning ourselves directly with format of the data sent in by the spotter. The **`Sighting`** class is very straightforward.

The code below shows part of the **`AnimalMonitor`** class that is used to aggregate the individual sightings into a list. At this point, all of the sightings from all the different spotters and areas are held together in a single collection. Among other things, the class contains methods to print the list, get the animals spotted, count the total number of sightings of a given animal, list all of the sightings by a particular spotter, remove records containing no sightings, and so on. All of these methods have been implemented using the techniques for basic list manipulation: iteration over the full list; processing selected elements of the list based on some condition (filtering); and removal of elements. The method signatures presented here are not complete for an application of this kind, but have been implemented to show examples of these different kinds of list processing.

```
package animalmonitoring.v1;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.Set;

/**
 * Monitor counts of different types of animal.
 * Sightings are recorded by spotters.
 *
 * @author David J. Barnes and Michael Kölling
 * @version 2016.02.29 (imperative)
 */
public class AnimalMonitor {
    // Records of all the sightings of animals.
    private List<Sighting> sightings = new ArrayList<>();

    /**
```

```
     * Add the sightings recorded in the given filename to the current list.
     * @param filename A CSV file of Sighting records.
     */
    public void addSightings(String filename) {}

    /**
     * Gets the set of all animals spotted.
     * @return Animals spotted.
     */
    public Set<String> getAnimals() {}

    /**
     * Print details of all the sightings.
     */
    public void printList() {}

    /**
     * Print the details of all the sightings of the given animal.
     * @param animal The type of animal.
     */
    public void printSightingsOf(String animal) {}

    /**
     * Print all the sightings by the given spotter.
     * @param spotter The ID of the spotter.
     */
    public void printSightingsBy(int spotter) {}

    /**
     * Print a list of the types of animal considered to be endangered.
     * @param animals A list of animals names.
     * @param dangerThreshold Counts less-than or equal-to to this level
     *                        are considered to be endangered.
     */
    public void printEndangered(List<String> animals, int threshold) {}

    /**
     * Return a count of the number of sightings of the given animal.
     * @param animal The type of animal.
     * @return The count of sightings of the given animal.
     */
    public int getCount(String animal) {}

    /**
     * Remove from the sightings list all of those records with
     * a count of zero.
     */
    public void removeZeroCounts() {}

    /**
     * Return a list of all sightings of the given type of animal
     * in a particular area.
```

```
     * @param animal The type of animal.
     * @param area The ID of the area.
     * @return A list of sightings.
     */
    public List<Sighting> getSightingsInArea(String animal, int area) {}

    /**
     * Return a list of all the sightings of the given animal.
     * @param animal The type of animal.
     * @return A list of all sightings of the given animal.
     */
    public List<Sighting> getSightingsOf(String animal) {}
}
```

This code will use the procedural collection processing techniques that we have already met. We will use this as a basis to gradually make changes to replace the list processing code with functional constructs, but first we'll complete the procedural code.

Exercise 1

Obviously the code above doesn't compile - there is no implementation for the methods. So go implement the methods.

The file *sightings.csv* contains a small sample of sighting records in comma-separated values (CSV) format. Pass the name of this file to the **`addSightings`** method of the **`AnimalMonitor`** object and then call **`printList`** to show details of the data that has been read.

Getting the file name right is actually trickier than it seems. We assume the project file hierarchy is as follows:

```
<run code from here>
├── build
|   └── classes
|       ├── animalmonitoring
|       |   ├── v1
|       |   |   ├── AnimalMonitor.class
|       |   |   ├── Main.class
|       |   |   ├── Sighting.class
|       |   |   └── SightingReader.class
└── src
    ├── animalmonitoring
    |   ├── v1
    |   |   ├── AnimalMonitor.java
    |   |   ├── Main.java
    |   |   ├── Sighting.java
    |   |   ├── SightingReader.java
    |   |   └── sightings.csv
```

If we use just **`addSightings("sightings.csv")`** the application won't be able to find the file; the file is in *src/animalmonitoring/v1/sightings.csv* relative to your project, but the JVM doesn't know that. We need to tell it exactly where to look. we can use **`System.getProperty("java.class.path")`** to get the absolute directory which contains the *.class* files, ie, *<path from root to project>/build/classes*. To get from here to the sightings file, we need to move up and down through the file hierarchy, something like **`addSightings(System.getProperty("java.class.path") + "/../../src/animalmonitoring/v1/sightings.csv")`**.

Exercise 2

You've probably already built a **`Main`** class to run the application. In the **`main`** method, develop code to write out something like the following

```
Pikachu is endangered: 14 left
Arceus is endangered: 24 left
Greninja is not endangered: 28 left
Mew is not endangered: 105 left
```

which displays for every animal whether or not it is endangered and the number of individuals remaining. In the above, the danger threshold was set to 25.

Exercise 3

Test your methods...!

Functional version

The animal monitoring project so far has introduced nothing new; it's just been an exercise in accessing collections. Now we're going to convert the procedural collection-processing to functional collection-processing. In your project, copy all the code from the **`animalmonitoring.v1`** package into a new **`animalmonitoring.v2`** package. Make sure it compiles and executes (You'll need to copy the *sightings.csv* file as well). From now we'll be working in the **`v2`** package.

In the functional style of collection processing, we do not retrieve each element to operate on it. Instead, we pass a code segment to the collection to be applied to each element. This makes our life easier. We can think about it like this: Imagine you need to give every child in a school class a haircut. In the old style, you say to the teacher: *Send me the first child*. Then you give the child a haircut. Then you say: *Send me the next child*. Another hair cutting session. And so it continues, until there are no more children left. This is the old style loop of next item/process.

In the new style, you do the following: You write down instructions for how to give a haircut, and then you give this to the teacher and say: *Do this to every child in your class*. (The teacher, here, represents the collection.) In fact, an interesting side-effect of this approach is that we don't even need to know how the teacher will complete the task. For instance, instead of cutting the hair themselves, they could decide to sub-contract the task to a separate person for each individual child, so that every child's hair is cut at the same time. The teacher would just pass on the instructions you gave them.

Your life is suddenly much easier. The teacher is doing much of the work for you. That is exactly what the new collections in Java 8 can do for you.

Collection `forEach` method

If we have a collection called myList, we can write

```
myList.forEach(. . . some code to apply to each element of list . . .)
```

The parameter to this method will be a lambda - a piece of code. The **`forEach`** method will then execute the lambda for each list element, passing each list element as a parameter to the lambda in turn.

Exercise 4

Rewrite the **`printList`** method in your version of the **`AnimalMonitor`** class to use a lambda function as an argument to **`sightings.forEach`**, something like

```
    public void printList() {
        sightings.forEach((...) -> {...});
    }
```

Exercise 5

Modify your **`printList`** method with the different syntax versions for the lambda:

```
(...) -> {...}
... -> {...}
(...) -> ...
... -> ...
```

Try out all the syntax variations for lambdas that we have shown. Recompile and test the project with each variation to check that you have the syntax correct. Test that **`printList`** still works and performs the same as before.

Benefit of lambdas

In practice, when we use lambdas, the lambda code is often short and simple, and the shortcut notation just discussed here can be used. This makes our code short and clear, and easy to write. Also, because we do not write the loop ourselves anymore, we are less likely to make errors - if we don't write the loop, it cannot be wrong.

There are, however, other benefits in addition to this. One is that this allows the use of streams, discussed in the next section. Another benefit is the use of concurrency (also called parallel programming). Most computers today have multiple cores and are capable of parallel processing. However, writing code that makes good use of these multiple cores is extremely difficult. Most of our programs will just use one core most of the time, and the rest of the available processing power is wasted. Learning to write code to use all cores is a very advanced subject, and many programmers never master it.

By using lambdas and moving the loop into the collection, we can now potentially use collections that handle the parallel processing for us. When we want parallelism, we just use an appropriate collection, and all the hard parallel processing code is hidden inside the collection class, written for us. Our code looks short and simple. Suddenly, using all your cores in your laptop becomes possible, with very little extra effort.

Streams and pipelines

We are now going to process the sightings using streams to filter, map and reduce the data.

Filter

The *filter* function takes a stream, selects some of it elements, and creates a new stream with only the selected elements. Some elements are filtered out. The result is a stream with fewer elements (a subset of the original).

Exercise 6

Modify the **`printSightingsOf`** method to replace the for loop with a pipeline of operations: **`stream`**, **`filter`** and **`forEach`** to print all the sightings of a given animal. As usual, test that it works as before.

Exercise 7

Write a method **`AnimalMonitor#printSightingsOn(int period)`** to print details of all the sightings recorded in a particular period, which is passed as a parameter to the method.

Exercise 8

Write a method **`AnimalMonitor#printSightingsOf(String animal, int period)`** that uses two **`filter`** calls to print details of all the sightings of a particular animal made in a particular period - the method takes the animal name and period as parameters.

Does the order of the two **`filter`** calls matter? Justify your answer.

Map

The *map* function takes a stream and creates a new stream, where each element of the original stream is mapped to a new, different element in the new stream. The new stream will have the same number of elements, but the type and content of each element can be different; it is derived in some way from the original element.

Exercise 9

Modify the **`printSightingsBy`** method to replace the for loop with a pipeline of operations: **`stream`**, **`filter`**, **`map`** and **`forEach`** to print all the sightings by a given spotter. But now, instead of invoking **`System.out.println(sighting.getDetails())`** we'll add a **`map`** operation to the stream after the **`filter`** to convert sightings into details. Then invoke **`System.out.println(details)`**.

Exercise 10

If a pipeline contains a **`filter`** operation and a **`map`** operation, does the order of the operations matter to the final result? Justify your answer

Exercise 11

Rewrite the **`printEndangered`** method in your project to use streams.

Reduce

The intermediate operations we have seen so far take a stream as input and output a stream. Sometimes, however, we need an operation that will "collapse" its input stream to just a single object or value, and this is the function of the *reduce* operation, which is a terminal operation. The reduce function takes a stream and collapses all elements into a single result. This could be done in different ways, for example by adding all elements together, or selecting just the smallest element from the stream. We start with a stream, and we end up with a single result.

In our animal monitoring project, this would be useful for counting how many elephants we have seen: Once we have a stream of all elephant sightings, we can use reduce to add them all up. (Remember: each sighting instance can be of multiple animals.)

Exercise 12

Rewrite your **`getCount`** method using streams, with **`reduce`** as the terminal operation.

Exercise 13

Add a method **`AnimalMonitor#getCount(String animal, int spotter, int period)`** that takes three parameters: animal, spotter ID, and period, and returns a count of how many sightings of the given animal were made by the spotter in a particular period.

Exercise 14

Add a method **`AnimalMonitor#getAnimals(int spotter, int period)`** that takes two parameters - spotter ID and period - and returns a **`String`** containing the names of the animals seen by the spotter in a particular period. You should include only animals whose sighting count is greater than zero, but don't worry about excluding duplicate names if multiple non-zero sighting records were made of a particular animal. Hint: The principles of using reduce with **`String`** elements and a **`String`** result are very similar to those when using integers. Decide on the correct identity and formulate a two-parameter lambda that combines the running "sum" with the next element of the stream.

Removing elements from a stream

We have noted that the filter operation does not actually change the underlying collection from which a stream was obtained. However, predicate lambdas make it relatively easy to remove all the items from a collection that match a particular condition.

Exercise 15

Rewrite the **`removeZeroCounts`** method using the **`removeIf`** method.

Exercise 16

Write a method **`removeSpotter`** that removes all records reported by a given spotter.