Functional Programming, Collections, and Pokemon

Joe Rossi

1 Problem Description

...world of Pokémon! My name is Oak! People call me the Pokémon Prof! This world is inhabited by creatures called Pokémon! For some people, Pokémon are pets. Others use them for fights. Myself... I study Pokémon as a profession.

On the desk there is my invention, the Pokédex! It automatically records data on Pokémon you've seen! It's a hi-tech encyclopedia! Only one problem though. It doesn't work! That's where you come in!

My last research assistant quit after he decided he wanted to be the very best, like no one ever was, blah blah. He wasn't so much interested in studying Pokémon. To catch them was his real quest, and he went on and on about it... What a worthless assistant!

You look like someone who can travel across the land, searching far and wide, without forgetting to do their job! I don't want a collection of lovable misfit creatures, I want data! And that's why I'm forbidding you from actually catching any Pokémon! You are a scientist after all!

1. **`MySortedSet:`** First things first! We need a data structure to store our Pokémon data. `MySortedSet` must implement the `FunctionalSortedSet` interface. There are a lot of methods to implement, but most of them are pretty simple. We've given you one of the complex ones, and you'll need understand how it works to write the rest (hint: see the `Stream` API for Java 8). Also, you should not implement any of the methods that have `default` implementations in their interfaces.

   This set implementation ensures that we don't get any duplicate elements, and that every element is added in sorted order. It also provides some filtering and sorting functionality for organizing your data.

   (a) A `MySortedSet` needs a backing `Collection`. Since we've seen how the `ArrayList` works in the `ArrayWrapper` homework, you **are** allowed to use an `ArrayList` here.

      i. This makes many of the required methods very simple to implement. Take advantage of the `ArrayList` API page!

   (b) A `MySortedSet` must also have a `Comparator` to determine the order elements are sorted. `Comparator` is a functional interface with a single method `compare`, that returns an `int`, similar to how `compareTo` in the `Comparable` interface works.

(c) We need to define one method from the FunctionalSortedSet interface using a lambda - `filter`.

(d) We need to define two methods from the SortedSet interface using lambdas - `subSet` and `tailSet`.

(e) Finally, we need to override `toString` from `java.lang.Object` to change the default `String` representation of a `MySortedSet` object to the `String` representation of each of its elements, each on its own line.

2. **`Pokemon`**: Before we write our Pokedox, we need some Pokemon to put in it.

   (a) Each Pokemon must have a name, number, and two elemental types (defined in the `PokemonType` enum)

   (b) You will also need to provide accessor methods for each of the 4 instance variables.

   (c) All `Pokemon` are `Comparable` by their numbers (in ascending order) by default.

   (d) Finally, you must override `toString` again to change the `String` representation of a `Pokemon` object. You should use `String::format` to create nicely aligned columns for number, name, primary type, and secondary type.

3. **`Pokedex:`** Now that we have the backing data structure for our Pokédex, it's time to write our `Pokedex` class! This class will be a wrapper for a `MySortedSet` and will provide some specific sorting and filtering options, specifically for `Pokemon`. To do so, you must implement the following methods using lambda expressions:

   (a) `listAlphabetically()`: reorders the `Pokemon` in the set alphabetically by name and puts them in a new set.

   (b) `groupByPrimaryType()`: reorders the `Pokemon`, grouping them by their primary types. (hint: try sorting by making a primary type comparison).

   (c) `listByType(PokemonType t)`: takes in a `PokemonType` (defined in the provided `enum`) and filters out any `Pokemon` that have both primary and secondary types different from `t`. So if either the primary or secondary type is `t`, the `Pokemon` should be in the set that is returned.

   (d) `listRange(int start, int end)`: this should filter the set to include all the `Pokemon` whose numbers are within the range [start, end], both inclusive.

2 Solution and Tips

When you run `KantoExpedition`, you should see something like this:

```
$ java KantoExpedition
Welcome to the world of Pokemon!

What do you want to do?
1: Walk around
2: Check Pokedex
3: Report to Professor Oak
4: Start over
5: Give up...
1
You wander around for a bit
.
..
...
....
.....
......
.......
........
.........
..........
...........
............
.............
..............
...............
................
Suddenly, a wild Omanyte appeared!
Praise be Lord Helix!
A new Pokedex entry was made for Omanyte!

Continue? (type n to stop walking):
You wander around for a bit
.
..
...
....
.....
......
.......
Suddenly, a wild Ryhorn appeared!
A new Pokedex entry was made for Ryhorn!

Continue? (type n to stop walking):
You wander around for a bit
.
..
...
....
.....
Suddenly, a wild Ekans appeared!
A new Pokedex entry was made for Ekans!

Continue? (type n to stop walking):
```

And then after a while:

```
1: Walk around
2: Check Pokedex
3: Report to Professor Oak
4: Start over
5: Give up...
2
What do you want to do?
1: List all (by number)
2: List all (alphabetically)
3: Group by Primary Type
4: List all of a single Type
5: List by Range of Numbers
3
#   20: Raticate    Primary Type: NORMAL     Secondary Type: NONE
#   36: Clefable    Primary Type: NORMAL     Secondary Type: NONE
#   52: Meowth      Primary Type: NORMAL     Secondary Type: NONE
#   53: Persian     Primary Type: NORMAL     Secondary Type: NONE
#   77: Ponyta      Primary Type: FIRE       Secondary Type: NONE
#  126: Magmar      Primary Type: FIRE       Secondary Type: NONE
#    9: Blastoise   Primary Type: WATER      Secondary Type: NONE
#   54: Psyduck     Primary Type: WATER      Secondary Type: NONE
#   91: Cloyster    Primary Type: WATER      Secondary Type: ICE
#  120: Staryu      Primary Type: WATER      Secondary Type: NONE
#  129: Magikarp    Primary Type: WATER      Secondary Type: NONE
#   66: Machop      Primary Type: FIGHTING   Secondary Type: NONE
#   24: Arbok       Primary Type: POISON     Secondary Type: NONE
#   31: Nidoqueen   Primary Type: POISON     Secondary Type: GROUND
#   41: Golbat      Primary Type: POISON     Secondary Type: FLYING
#  109: Koffing     Primary Type: POISON     Secondary Type: NONE
#  112: Rydon       Primary Type: GROUND     Secondary Type: ROCK
#   49: Weedle      Primary Type: BUG        Secondary Type: POISON
#   74: Geodude     Primary Type: ROCK       Secondary Type: GROUND
#   75: Graveler    Primary Type: ROCK       Secondary Type: GROUND
#  138: Omanyte     Primary Type: ROCK       Secondary Type: WATER

What do you want to do?
1: Walk around
2: Check Pokedex
3: Report to Professor Oak
4: Start over
5: Give up...
2
What do you want to do?
1: List all (by number)
2: List all (alphabetically)
3: Group by Primary Type
4: List all of a single Type
5: List by Range of Numbers
4
Enter a type (or 'help' for a list of valid types)
rock

#   74: Geodude     Primary Type: ROCK       Secondary Type: GROUND
#   75: Graveler    Primary Type: ROCK       Secondary Type: GROUND
#  112: Rydon       Primary Type: GROUND     Secondary Type: ROCK
#  138: Omanyte     Primary Type: ROCK       Secondary Type: WATER
```

The provided files contain more details about what you'll need to implement and how you need to do it. Do not modify `PokemonPopulation`, `KantoExpedition`, or the `FunctionalSortedSet` interface, nor the functional methods in `MySortedSet`.

It is absolutely vital that your submission compiles with the provided source files. Non-compiling solutions will be given a zero: no exceptions.

3 Javadocs

We are going to have you do Javadocs for this assignment (and for all assignments here on out). Javadocs are a clean and useful way to document your code's functionality. For more information on what Javadocs are and why they are awesome, the online documentation for them is very detailed and helpful. The relevant tags that you need to have are `@author`, `@version`, `@param`, and `@return`. Here is an example of a properly Javadoc'd class:

```
import java.util.Scanner;

/**
 * This class represents a Dog object.
 * @author George P. Burdell
 * @version 13.31
 */
public class Dog {

   /**
    * Creates an awesome dog (NOT a dawg!)
    */
   public Dog(){
      ...
   }

   /**
    * This method takes in two ints and returns their sum
    * @param a first number
    * @param b second number
    * @return sum of a and b
    */
   public int add(int a, int b){
      ...
   }
}
```

4 Checkstyle

You must run checkstyle on your submission. The checkstyle cap for this assignment is **100** points.
Review the Style Guide and download the Checkstyle jar and associated XML file. Run Checkstyle on your code like so:

```
$ java -jar checkstyle-5.6-all.jar -c cs1331-checkstyle.xml *.java
Starting audit...
Audit done.
```

The message above means there were no Checkstyle errors. You can easily count Checkstyle errors by piping the output of Checkstyle through `wc -l` and subtracting 2 for the two non-error lines printed above (which is how we will deduct points). For example:

```
$ java -jar checkstyle-5.6-all.jar -c cs1331-checkstyle.xml *.java | wc -l
      2
```

Alternatively, if you are on Windows, you can use the following instead:

```
C:\> java -jar checkstyle-5.6-all.jar -c cs1331-checkstyle.xml *.java | findstr /v "Starting audit..." |
     findstr /v "Audit done" | find /c /v "hashcode()"
0
```

> Food for thought: is there a one-liner like above that shows you only the number of errors? Hint: `man grep`.

The Java source files we provide contain no Checkstyle errors. For this assignment, there will be a maximum of **100** points lost due to Checkstyle errors (1 point per error). In future homeworks we will be increasing this cap, so get into the habit of fixing these style errors early!

5 Turn-in Procedure

Submit all of the Java source files you modified and resources your program requires to run to T-Square. Do not submit any compiled bytecode (`.class` files), the Checkstyle jar file, or the `cs1331-checkstyle.xml` file. When you're ready, double-check that you have submitted and not just saved a draft.

Please remember to run your code with Checkstyle!

Verify the Success of Your Submission to T-Square
Practice safe submission! Verify that your HW files were truly submitted correctly, the upload was successful, and that the files compile and run. It is solely your responsibility to turn in your homework and practice this safe submission safeguard.

1. After uploading the files to T-Square you should receive an email from T-Square listing the names of the files that were uploaded and received. If you do not get the confirmation email almost immediately, something is wrong with your HW submission and/or your email. Even receiving the email does not guarantee that you turned in exactly what you intended.
2. After submitting the files to T-Square, return to the Assignment menu option and this homework. It should show the submitted files.
3. Download copies of your submitted files from the T-Square Assignment page placing them in a new folder.
4. Recompile and test those exact files.
5. This helps guard against a few things.
   (a) It helps insure that you turn in the correct files.
   (b) It helps you realize if you omit a file or files. [1] (If you do discover that you omitted a file, submit all of your files again, not just the missing one.)
   (c) Helps find last minute causes of files not compiling and/or running.

[1] Missing files will not be given any credit, and non-compiling homework solutions will receive few to zero points. Also recall that late homework will not be accepted regardless of excuse. Treat the due date with respect. The real due date is midnight. Do not wait until the last minute!