Match-IT Labs

November 2018

1 Lab 1 - using abstract data types

Goal: You will practice implementing algorithms using the abstract data types list, *Set* and *Map*. You will also use interfaces and classes from the Java Collection Framework, which both describes and implements these abstract data types.

During the lab, you will see a lot of new classes. The idea is that you should get a taste of how the Java Collection Framework class library can be used. During the course we will then look at each abstract data type and discuss how the corresponding classes in JFC look under the hood and how to use them.

1.1 Preparations

Answer the questions using the course literature, the lecture slides and / or the documentation for each interface.

F1 Look at the following code snippet:

```
List<Integer> nbrs = new ArrayList<Integer>();
for (int i = 0; i < 100; i += 10) {
    nbrs.add(i);
    nbrs.add(i); // Note, the number is added twice
}
for (int a : nbrs) {
    System.out.println(a);
}
```

(a) How many rows are printed?

(b) Can we be sure of the order of the items?

F2 Assume we switch out the first line with the following:

```
Set<Integer> nbrs = new HashSet<Integer>();
```

(a) How many rows are printed now?

(b) Can we be sure of the order of the items?

F3 Look at the following snippet:

```
Map<__________, __________> m = new HashMap<__________, __________>();
m.put("albatross", 12);
m.put("pelikan", 27);
m.put("lunnefagel", 19);
m.put("albatross", 7);
System.out.println(m.get("albatross"));
```

Fill in typing arguments on the dashed lines above, so that the types match the code. (You may not use the *Object* type here.)

(a) What is printed?

F4 In the interface *Map* there is a method used for testing if a key is present. What method is it?

F5 Read through chapter Background

F6 Solve exercises Exercises, 2, 3, 4

F7 Read the remaining exercises

1.2 Background

Historians and literature scientists sometimes use computer-based methods to get general information about large texts. For example, by calculating the number of occurrences of place names (and other alike), one can get an idea about the geography thats described in the text

With this method, we can examine Selma Lagerlöf's book *Nils Holger's wonderful trip through Sweden* and count the number of occurences of the landscape names. That way we can understand something about the image of Sweden that is portrayed in the book. Lagerlöf's book was published 1906-1907, under the national-romantic the period, when many of the 20'th century conceptions about the Swedish nation were formed.

When we examine a text, such as Lagerlöf's book, we count occurences of some words, such as names places. In this task you will implement classes to count words in different ways. In particular, we will focus on which abstract data types you can use.

1.3 Exercises

D1 Start by acquainting yourself with the *TextProcessor* interface. It contains methods for processing read text, one word at a time, and methods for presenting a result. We will use this interface to handle different types of text analysis methods in a consistent way.

```
public interface TextProcessor {
/**
* Called when a word is read.
* The method will then update the statistics.
*/
void process(String w);
/**
* Method called when all the words in the sequence are read.
* The method should print a readable representation of the statistics.
*/
void report();
}
```

(In this case we could also use an abstract class, but the interface fits better because *TextProcessor* does not have any own attributes or methods.)

D2 In the project there is also a class *SingleWordCounter*, which implements the interface above. This class is used to count how many times a single word occurs. The class contains an error, which makes that the number is always 0 (zero). Find the error and fix it.

D3 The project also contains a program *Holgersson.java*. There, a *SingleWordCounter* instance is created, which will count the number of occurrences of the word "nils". We spell the name with lowercase because the *Holgersson* program converts all words into lowercase upon loading (using the *String.toLowerCase* method).

Then the book is read. The text of the book is saved in the text file nilsholg.txt, and starts with a pair of poems before chapter 1. You can open the file and see what it looks like.

All the words in the file are processed and our *SingleWordCounter* is updated. Finally the result is printed.

Run the program and check that the printed result is correct. The name "Nils" occurs 75 times in Lagerlöf's novel. If your result fails, it may be due an error in the previous task.

Comment to the line *s.useDelimiter(...)*: here the Scanner object is configured so that the separator characters ,.:!!?"" are filtered from the read words. The call looks a little strange, because we used a *regular expression* to indicate that all of these characters should be considered as separator characters. You do not have to worry about how this line works.

D4 Before we continue, we want to extend the *Holgersson* program so that it can handle several *TextProcessor* implementations on the same text.

Modify the program so that it has a list of *TextProcessor* objects. To begin with the list should only contain the TexProcessor instance from before (the one that counts "Nils").

Each time a word is loaded, all *TextProcessor* objects in the list will get their *process* method called. After all text has been processed, all *TextProcessor* objects must print their respective results.

Add a line to your *Holgersson* program, so that even the number of occurrences of the word "Norway" counts. Your list should therefore contain two *TextProcessor* objects, and you will get the following results:

```
nils: 75
norge: 1
```

D5 So far, we have had to create a new object for every word we count. Now we want to introduce a new type of text analysis, where not only a single word counts, but several. We will count how many times the different Swedish landscapes are mentioned in the book.

Create a new class *MultiWordCounter*, which implements the *TextProcessor* interface and works as follows:

- The constructor should take a vector of strings as a parameter. The vector contains the words we want to count. The following example shows how such a constructor should work:

```
String[] landscapes = { "blekinge", "bohuslan" /* , ... */ };
TextProcessor r = new MultiWordCounter(landscapes);
```

- Your class *MultiWordCounter* should have exactly **one** attribute, and that attribute should be off type *Map* (with appropriate type arguments - compare to the exercises in Preparations). This *Map* attribute is used to keep track of how many times the searched words (landscape name in the example above) occur. Initially, this *Map* contains the value 0 (zero) for each searched word (landscape name). The method *process* increases the number **if** the given word is one of the searched words.
- Even though the attribute has the type of *Map*, the actual object to be stored in it should be an instance of the concrete class *HashMap*. In this way, for as long as possible, your program becomes independent of which implementation of the *Map* interface is actually used. Thus, the name *HashMap* should only occur in one place in the class (not counting import statements).
- The *report* method should print all the keys and their respective values in your *Map*. **Tip:** You can go through all the keys in a *Map* like this (if we assume m is a *Map* whose keys are of type *String*):

```
for (String key : m.keySet()) {
    // Do something with key and m.get(key)
}
```

D6 Add a *MultiWordCounter* object for the landscapes in your list in the *Holgersson* program. Note that there is a useful string vector present in the program.

Run the program. In the result, we see that the border areas (Skåne, Lappland) are mentioned relatively often. Perhaps it was important to show that Sweden was still quite large, despite Norway leaving union the year before? (Norway is mentioned only once.)

D7 We will now retrieve the information about the book in another way. By counting all the words, not just landscapes, we can get an idea about the book's content. However, we must exclude some common words, such as "och", "ett" and "att", to get a meaningful result. Again, we need a table of words, but now to count all words *except* for a few chosen ones.

Create a class *GeneralWordCounter*, which implements the *TextProcessor* interface and works as follows:

- The constructor should take a *Set* as a parameter. This *Set* contains exception-words read from the undantagsord.txt file:

```
Scanner scan = new Scanner(new File("undantagsord.txt"));
Set<String> stopwords = ...; // a suitable set is instantiated
... ... // words are read from the scanner 'scan'
... ... // and saved in the set
TextProcessor r = new GeneralWordCounter(stopwords);
```

  The file undantagsord.txt file is in your Eclipse project. You can open it and see what it looks like.

- Use a *Map* to keep track of how many times the respective words occur.
- Use *HashMap* in this class as well, but like before, the name *HashMap* should only be used once (not counting import statements). You can choose which implementation of the *Set* interface you want to use yourself.
- The method *process* counts all words unless they are in the *Set* of the exception words. The first time a new word is detected, its added with the number number of occurences equal to 1, and the following times the same word is detected, that number is incremented.
- The *report* method should print all words that occur 200 times or more.

D8 Add a *GeneralWordCounter* object to your list in the *Holgersson* program. Run the program. What are the most common words in Lagerlöf's book?

The interest in the border regions is also shown indirectly here. Ledargåsen Akka is named after a Lappland mountain massif, and the boy comes from Skåne.

D9 When you implemented the method *report* above, we set the limit at 200 words. Such a fix limit is unsatisfactory: for a longer book, the printout may be unmanageably long, and if we analyze a shorter text, we may not get any results at all. Instead, it would be better to print, say the five most common words.

For this we will use Javas embedded sorting algorithm for lists. Its an effective algorithm, which allows us to provide a custom order for the elements. This comparison is described using a particular interface, and in this and the following two exercises you will see how it works.

First, we need a list of words with their associated occurence numbers. There is no way to get such a list from our *Map*, but we can get a *Set* of such word-number pairs with the following method:

```
/** Returns a Set view of the mappings contained in this \textit{Map}. */
Set<Map.Entry<K,V>> entrySet();
```

We can then get a *Set* of objects of type *Map.Entry¡String, Integer¿*. Each such object contains a word and its associated number. When we have such a *Set*, we can place the elements in a list by passing the *Set* as a parameter to the list's constructor.

- Add the following lines in the *GeneralWordCounter* class, in the *report* method. Remove (or comment out) the code that was there before.

```
Set<Map.Entry<String, Integer>> wordSet = counts.entrySet();
List<Map.Entry<String, Integer>> wordList = new ArrayList<>(wordSet);
```

- Modify *report* (a few lines) so that five first items of the list (*wordList*) are printed out.

Test the program and make sure that you get five lines in the printout from *GeneralWordCounter*. These five lines are not the five most common words, but only the five words that happened to be first in the *Map* of whichever implementation you chose.

(The list is not yet sorted in the correct order, so this is just an intermediate step, where we verify that the step from *Map* to *wordList* works.)

D10 Now we will use Java's built-in sorting algorithm. We will do it by making small changes step by step. In several of the steps, compilation errors will occur (which will be then solved in the next step).

Try to understand what the compilation errors mean! You can ask your supervisor if you're confused.

- Create a new, empty class in Eclipse. Name the class *WordCountComparator*.
- Add the following lines above the lines above:

```
wordList.sort(new WordCountComparator());
```

We therefore state that a *WordCountComparator* object should be used to decide the order. We will shortly get to how it works.

You get a compilation error. Why?

- As stated in the compilation error, our class *WordCountComparator* needs to implement interface *Comparator*. Change in the new *WordCountComparator* class, so that it starts like this:

```
package textproc;

import java.util.*;

public class WordCountComparator implements Comparator<Map.Entry<String,Integer>>
```

We thus add an interface for the class (and the associated import statement). Thus we indicate that *WordCountComparator* can be used for the kind of comparisons as the sorting requires. In other words, we promise, that the class has a method *compare*, which is required by the sorting algorithm. (The method is called *compare* because it is named so in the *Comparator* interface.)

The type argument above, *Map.Entry¡String, Integer¿*, means our class *WordCountComparator* can compare precisely such *Map.Entry* objects as we have in our program.

The compilation error for *sort* disappears. Instead, we get a new compilation error in the class *WordCountComparator*, as it does not yet implement the comparison. ("WordCountComparator must implement method ..."). The class does not yet fulfill the promise.

- Hold the mouse pointer over the compilation error in *WordCountComparator* and select "Add unimplemented methods" in the small "quick fix" list that appears (see Figure 1.3). You will now get an auto-generated comparison method for the comparison. Eclipse has of course no idea what order we want, but the method compiles without any problems.

Test the program. The order is still wrong, but now only a few code lines remain before we get a sorted printout.

D11 In the *compare* method, the two parameters, *o1* and *o2*, are to be compared. Javas sorting algorithm calls the method repeatedly for two word-number pairs, and sorts the elements based on the order that our *compare*-method indicates.

The method should return a value according to the following patterns:

- greater than 0 if the first element is "bigger" than the other
- less than 0 if the first element is "smaller" than the other
- exactly 0 if the two are to be considered "equal".

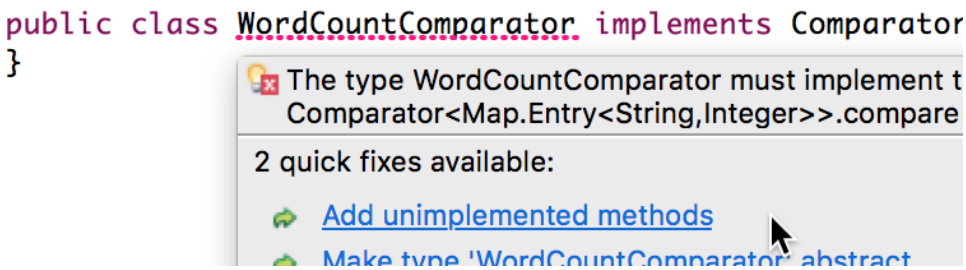

Figure 1: A compile error with a quick-fix suggestions in Eclipse

Here, "smaller" and "equal" refer to the desired order. You surely recognize the idea from the method *compareTo* in the *String* class.

Implement the method *compare* so that the elements are sorted based on the number of occurrences, in descending order. Run the program and check the results.

Tip: Do you wonder how to get the integer from a *Map.Entry* object? Please look at the specification. Remember, there are a pairs come from a *Map*, where the keys are strings and the values integers.

D12 Adjust your *compare* method (from task D11) so that the elements are first sorted by the number of occurrences, and then in alphabetical order.

Test! First increase the number of words printed in the *report* method in *GeneralWordCounter* so that you see that your change in *compare* does matter.

D13 You can measure the execution time of a code section using *System.nanoTime* This method returns the number of nanoseconds that have elapsed since some unspecified time point. By subtracting two such times you measure the time passed, for example in milliseconds:

```
long t0 = System.nanoTime();
... // Code that we want to time
long t1 = System.nanoTime();
System.out.println("Time: " + (t1 - t0) / 1000000.0 + " ms");
```

Adjust the *Holgersson* program to print the time of the program, as above. Run the program three times and select the median value of execution times (the middle value). Remember this median value.

D14 Adjust your classes *GeneralWordCounter* and *MultiWordCounter*, so that it uses *TreeMap* instead of *HashMap*. If you've done everything right, it's enough to change the code in only one place.

- Does your program still work?
- How is the order affected in the printed result?
- How is execution time affected? (Also calculate the median value from three runs.)

D15 Think about the following, and discuss with your supervisor:

- What is the difference between *Map* and *HashMap*?
- What is the difference between *HashMap* and *TreeMap*? What is the reason for these differences?
- In the laboratory you have used the abstract data types *Set* and *Map*. What special properties do they have that make them suitable to use for the purposes of this lab?
- You have also used Java's built-in sorting using a sort order that you yourself decided. How does it work? And what function does the *Comparator* interface have in this?