# CS 455 Lab 10: Java Map, Java sort, callbacks

Spring 2019 [Bono]

---

## Goals and Background

In this lab you'll add some functionality to the concordance program discussed in the 3/21 lecture. You'll get a chance to try out some of the Java and Java library features we've been covering in lecture and the readings over the last few weeks, including Maps, sorting, and interfaces. You'll also run the program on some real data, and use the results to learn more about the data files.

By the way, in this version of the program we have added a `filter` function that strips a word of punctuation and converts it to all lower case, so, for example, "Yikes", "yikes", and "Yikes!!!" would be counted as instances of the same word. This makes the concordance results more meaningful when run on files of real English text.

## Reading and Reference Material

- Horstmann, Sect. 14.8, on Sorting in the Java Library, including Special Topic 14.4 on the Comparator interface.
- Horstmann, Sect. 15.4 on Maps.
- Horstmann, Ch. 10 Interfaces, especially section 10.4 on Callbacks
- Java API documentation pages.
- 2/28 lecture example using Comparable and Comparator to sort Student objects (available on Vocareum)

## Exercise 1 (1 checkoff point)

Review the code in `Concord.java`, which is a more complete version of what we did in lecture, and then compile and run the program to see what it does currently. Note: the main program is in `ConcordDriver.java`. Use input and output redirection so you can save your data files and corresponding output. Once you run it on a small file that you create, run it again on one of the story files that came with the lab (`poe.txt` or `melville.txt`), redirecting the output to a file.

Implement the method `printSorted` in the `Concord` class. (We've already added the header for you.) This function will print out a `Concord` object (a map of words and their frequencies) sorted in decreasing order by frequency.

For example, if the current program printed:

```
Concord alpha order:
and 3
be 1
better 1
dog 6
i 5
the 12
```

The new version would print:

```
Concord numeric order:
the 12
dog 6
i 5
and 3
```

```
be 1
better 1
```

The new version of the program should not print out both copies of the concordance, just the one ordered by frequency. (You just need to change which lines of code are commented out in ConcordDriver to call your new method.) The rest of this Exercise description gives information on how to solve this problem (with one question to answer along the way).

There is more than one way to proceed, but there are a few issues to keep in mind. The Java `sort` method does not work on a `Map` directly. Also, we have to make sure that the Java `sort` method we use will sort items in the correct order, and on the correct field.

---

**Question 1.1** The TreeMap inside Concord keeps the entries in order. Why can't we just use its iterator or a for-each loop to visit the values in order? (Hint: there's iterator code like this in the `print` method of `Concord`.)

---

The most straightforward way to solve the problem is to copy the entries into an ArrayList, sort the resulting ArrayList, and then print out the resulting ArrayList. Hint: There are methods to support doing the copy as one operation (i.e., no loop necessary).

Because the element type for our ArrayList was not defined by us, we can't modify it to make it `Comparable` and implement `compareTo`. Also, because we are sorting in decreasing order in this case, it's unlikely that an existing `compareTo` would order things the way we want here. We need to create a separate comparison function to compare two elements of our ArrayList, and have the sort function use that. There is a two-parameter version of the sort methods that allow us to do just that. The comparison function needs to be in a class that implements `Comparator`.

`Comparator` is an example of an interface that contains just one function, such that code that calls the function (e.g., the sort method calls it) can be customized by substituting different Comparator implementations. This general idea is called a *callback* and was discussed in more detail in Section 10.4 of the text. The Comparator interface, in particular, was discussed more thoroughly in Special Topic 14.4 of the text.

We also provided an example of using this in the example code from a recent lecture (see Reading and Reference material above for more information.)

It's fine if you put your Comparator implementation in the Concord.java file (you are allowed to have two classes in one file); only the one that matches the file name prefix will have the `public` label.

The naming of these two similar interfaces is confusing at first, but it actually makes sense. A Comparable is an object that can be compared to other objects of the same type (using a compareTo method of the object). A Comparator is an object whose only purpose is to compare two other objects.

You will not have to write very much code to solve the problem: the challenge will be figuring out the correct code to write. You may need to refer to the online Java API documentation (linked above) to figure out the details.

Make a small test file and show the TA your program running on that file and also show him the code you wrote. Also show him the answer to question 1.1.

## Exercise 2 (1 checkoff point)

Take quick look at the contents of the two test files given: `poe.txt` and `melville.txt` (you do not have to read the whole text). Run your resulting program on each of these files. Use Unix output redirection to save the results of each of these runs (call them `poe.out` and `melville.out`).

**Question 2.1** Report to the TA five of the most frequently occurring words in the English language based on what you find in the results. (It doesn't have to be the absolute top five words.)

**Question 2.2** For each file report to the TA the most frequently occurring word in that file that is *not* one of the most frequently occurring words in the English language. For each of your answers, tell why it occurs so frequently in this file.

**Question 2.3** In what order do words that have the *same* frequency appear in your output? (Look at the last part of your output file to see many examples of this.) Why? (Hint: look at the documentation of the sort method you called.) Would they be in a different order if we had used a HashMap?

## Exercise 3 (1 checkoff point)

Recall that `Comparator` is an example of a general idea called a callback (another term used is *function object*). In this exercise we'll do a second callback example.

We have written our own such callback `interface` called `Predicate` that has a method whose parameter is a Map entry, and return type is `boolean`. A class that implements this interface would tell us whether a given entry satisfies some condition. See the code in `Predicate.java` for its definition. Note: "predicate" is another word for a boolean function.

Note: For this exercise we're using a different version of the main program class called `ConcordDriver3.java`. The new version of main takes a command-line argument we'll discuss soon.

We have also written a `Concord` method, `printSatisfying` that uses the Predicate interface to print out all entries that satisfy a given condition. Take a look at that method now. This method can be customized to do many different things, depending on what exact Predicate object type you pass to it. For instance you could use it to print out all words that occurred exactly 10 times, or all words ending in "ing". For each of these customizations, you would need to create a different class that implements the Predicate interface and that has the code to do the exact test you wish.

In this exercise you are going to use `printSatisfying` to print out only the concordance entries whose words are of at least some specified length. The entries will be printed out in Map order (i.e., this exercise does not involve sorting the entries). The threshold length to use will be given as a command-line argument (the code to get the argument is already in ConcordDriver3.java).

For example, with the input used in the example in Exercise 1, and the command-line argument `3` (i.e., it will only list entries whose words have at least 3 characters), the output would be:

```
and 3
better 1
dog 6
the 12
```

To do this, you'll need to create your own implementation of the Predicate interface. For your Predicate to handle different thresholds, it will need a constructor that takes the threshold as an argument and saves it in the object that has the the callback method.

Suppose your `Predicate` implementation is a class called `LargeWordPred`. Here is an example of how we might use it to control which concordance entries get printed (Note: pay attention to two possible ways we can do the call below):

```
  Predicate testLarge = new LargeWordPred(10);

  // prints all words of length >= 10
  // (uses temp var)
  concord.printSatisfying(System.out, testLarge);
```

```
// prints all words of length >= 12
// (uses anonymous object)
concord.printSatisfying(System.out, new LargeWordPred(12));
```

---

**Question 3.1** Why don't we need to save the `LargeWordPred` object in a variable (second call above)?

---

Write the class (you can put it in `ConcordDriver3.java`), and then add the necessary code in `main` to use it to print out the entries with length at least as large as the threshold given on the command line.

Run your program with different values for the threshold until you get a very small amount of output. Save your output in files labeled by the threshold used, e..g., `melville.10out` would hold all entries with words of length 10 or greater from the Melville text.

---

**Question 3.2** Using your results from your code and experiments above, answer the following questions: What are the longest non-hyphenated words occurring in each of `melville.txt` and `poe.txt`, how long are they, and how many times did each of them occur in the original file? (Note: there may be multiple words that are the longest.)

---

Show the TA your code and the output files that helped you to answer the question. (DEN students, list the names of these output files in your README.)

## Checkoff for DEN students

Make sure you put your name and NetID in the source code files and README.

When you click the `Submit` button, it will be looking for and compiling (for source code) the files `README`, `poe.out`, `melville.out`, `Concord.java`, and `ConcordDriver3.java`.

In addition to the files listed above, the Lab TA will be looking for the files you mentioned in your README that helped you to answer Question 3.2.

---