Lab 6, CSC/CPE 203
Sorting, comparisons and lambda
Due: 11/ 7@10PM (No Demo)

Sorting/ordering data is a common technique used in computing applications and, as such, Java has built-in support to allow various methods to compare data (which can be used for sorting). In addition, a closely related task, of specifying a small block of code (computation) that can be passed around and executed later, is supported in Java 8 with the added support for *lambda expressions*. (Historically, there was some non-syntactic support in various libraries even earlier than this). The inclusion of *lambda expressions* can aid the programmer with common tasks (with comparisons being the first example we will tackle in this lab). At the surface level, a lambda expression can be seen as a syntactic simplification of a common use of anonymous inner classes (or even plain classes). Though this syntactic reduction is of great value, lambda expressions, coupled with additional support throughout the Java libraries (in particular, the `java.util.function` and `java.util.stream` packages), provide another means for designing programming solutions. You saw the examples on functional interfaces and we will discuss stream later in the course.

This lab introduces the mechanics of lambda expressions as motivated by the common task of comparing two objects. Later labs will revisit the use of lambda expressions in different contexts.

Objectives

- To explore the task of comparing data using various Java mechanism, including comparators and lambdas
- To practice using lambda expressions for comparisons to sort data
- To practice understanding the application of lambda expressions via reading and understanding code examples using lambdas more generally (then just for sorting)

Given Files

Retrieve the files provided for this lab from Canvas. This Lab has two parts. Start with Exercise folder.

Lambda Expression — Comprehension Exercises

Examine the provided testing file in the `exercises` directory. Each test case includes some comments about the intended behavior. Edit the test cases so that they all pass. Go through each exercise test to figure out what is correct expected value is.

Note that you should read the code to determine the correct expected value for each test case. Running the tests and then just plugging the computed values back in will not improve your understanding of lambda expressions. You will want to solidify your understanding since we will continue to explore this feature for the remainder of the course and because you will use lambda expressions in Java code beyond your academic career.

Comparators and Lambda Expressions

Examine the provided files in the `comparator` directory.

For this part, we will compare different implementations of the `java.util.Comparator` interface, including the use of lambda expressions. A `Comparator` allows, via the `compare` method, one to compare two objects to determine which "comes before" the other. For example, one might define a `Comparator<Integer>` to determine which of two `Integer` objects comes first by ascending order.

Various methods take `Comparator` objects to generalize algorithms (such as sorting). The benefit is that passing different `Comparator`s allows for changing the order. Use of the `Comparator` interface (especially as opposed to the `Comparable` interface) allows one to remove the "ordering" logic from the objects to be compared. Instead, this logic is placed elsewhere to allow for multiple "orderings" of the same data.

The `Comparator` interface requires its implementing classes to implement a `compare` method that takes two arguments and returns an `int` value indicating the relative order of the arguments. **A negative return value indicates that the first parameter object "comes before" the second; a positive return value indicates that the second parameter object "comes before" the first; and zero indicates that the values are equivalent by the ordering.**

ArtistComparator

Define the `ArtistComparator` class (yes, for this part, as a class), implementing `Comparator<Song>`, to compare two `Song` objects and order them by artist (in ascending order). The `Song` class is in the provided code.

Write a few tests of your `ArtistComparator` in the provided `TestCases.java` by comparing two `Song` objects (you may use elements of the `songs` array).

For example, when you compare the first and second songs in the song list, the result should be less than 0, because the "Decemberist" is lexically before "Rogue Wave". Write another test case to test for alternative cases. Note, counting like a human, not a computer, so the first song would be song at position zero.

Title Comparator — As a Lambda Expression

Functional interfaces (those that declare only a single required method) can be "implemented" by lambda expressions. A lambda expression is a stand-alone, anonymous function (in Java, they turn out to be shorthand for anonymous inner classes).

For this part, you *will not* define a new class. Instead, in `TestCases.java`, assign a lambda expression to a `Comparator<Song>` variable local to your testing method. This lambda expression should act as a comparator on `Song` objects that orders them in ascending order by title.

Write a few tests to verify that this comparator works. For the test cases, you only need to compare two songs at a time (but consider writing more than one comparison to accurately test your implementation).

For example, when you compare the first and second songs in the song list, the result should now be greater than 0, because the Decemberist's song "The Mariner's Revenge Song" is not lexically before Rogue Wave's "Love's Lost Guarantee".

Year Comparator — Using a Key Extractor

If you examine the Javadoc for `java.util.Comparator` interface, you will notice that there are many more methods than `compare`, which can be useful. Of note are the many static methods that can be used to create `Comparators` based on a "key extractor" function.

Using a key extractor instead of a lambda expression is less general, because you can only write a reference to an existing method. It can be slightly more compact, and some find this style to be more readable. It is unlikely that there will be any notable performance difference between the two styles. Buried in the sea of dross and questionable advice that one can find on stack overflow, some insight into why this duplicative language feature was added, is given by **Brian Goetz**, one of the architects of this part of the Java Language, in this post.

Write a few test cases test a `Comparator<Song>` ordering by year, in *descending* order (in other words most recent songs would be listed first -- getting this ordering proper may require a bit of research). But for this part, you must use an appropriate `static comparing` method by providing a "key extractor" function.

For example, when you compare the second and third songs in the song list, the result should now be greater than 0, because Rogue Wave's "Love's List Guarentee" is from 2005 while the Avett Brother's "Talk on Indolence" is from 2006 and we are comparing in descending order.

Comparator Composition

The comparators defined thus far compare only a single field to determine an ordering, but it is often the case that when trying to order two objects one might want to order first by a primary key and then, if the primary key matches, by a secondary key. For instance, one might wish to order songs by artist and by year for songs by the same artist.

For this part, you will define the `ComposedComparator` class, implementing `Comparator<Song>` (this class should be generic, but for now we will fix it to `Song` objects). This comparator must define a constructor that takes two `Comparator<Song>` objects, `c1` and `c2`. The `compare` method must be defined to use `c1` to compare the `Song` objects and then, if they are equivalent by the `c1` ordering, use `c2`.

Write a test using this comparator; be sure to select a pair of songs that demonstrate the sequencing behavior of this comparator. (Hint: First sort based on their Year)

For example, when you compare the fourth and eighth songs in the song list, they are both by the same artist, but with different years. When compared think about what the result would be based on the years of these songs.

thenComparing

Composition, or sequencing, of the sort in the previous part is a relatively common technique. As such, the `Comparator` interface actually supports this via a (default) method named `thenComparing`. On an existing comparator object, one can call `thenComparing` and pass to it the next comparator to use in the sequence.

Declare a `Comparator<Song>` variable in a testing method and initialize it with a lambda expression (or using a key extractor) comparing songs **by title.** Then invoke `thenComparing` on this object passing to this method another comparator (or a key extractor) that will compare songs **by artist.** Write a test to verify that this comparator works as expected (i.e., orders by title and by artist when the titles match).

For example, when you compare the fourth and sixth songs in the song list, they are both named "Baker Street", however they are by two different artists, Gerry Rafferty and the Foo Fighters. Thus, the result of a comparison should be greater than 0, because 'F' comes before 'G'.

sort

Using the technique in the last part (lambda expressions with `thenComparing`), complete the sorting test by passing a comparator that orders **by artist, then title, and then year** (each in **ascending order**).

For this test case, there is a correctly ordered song list to use for your comparison.

Submission

Submit all your .java files in the Canvas. No demo.