Objective. To gain experience using Comparators and *lambdas* to specify sorting and filtering criteria.

Discussion. This week we'll extend the `Vector` class in the `structure5` package to include a sort method and a filter method. The result will be called a *SortingVector*. In most ways, a `SortingVector` acts just like a `Vector`. The additional sort method will allow us to order data *in-place*. For the most flexibility, we'll allow the user of the sort method to specify a *comparator*—an object that simply wraps a function `compare(a,b)` that can be used to impose relative order on the pairs of objects (a and b) encountered. The filter method will allow us to construct a *new* vector that contains only the objects that meet a particular condition. The condition is determined by a *predicate*—again, an object that simply wraps the function `test(item)` that is used to evaluate whether *item* should be copied to the new vector or filtered out.

To test our new, more powerful *SortingVector*, we'd like to write a few applications that allow us to answer questions about objects stored in a database. In particular, we have downloaded a database of pet registrations from Seattle\(^1\). The database is stored in a simple *comma-separated values* (CSV) format where each line describes one registered pet. (You may have seen these CSV files before: they're a common format for storing the rows of data from a spreadsheet.) Internally, each pet will be described by an object from the Pet class. We’ll read the pet descriptions from the database and build a vector of Pets. We'll design the Pet class to store data about a particular pet, but they won't have any “natural” ordering; we'll sort and filter Pet objects based on code we write later. We have several (silly) questions we'd like to ask about these pets; we think they can all be answered by a combination of sort and filter operations that perform question-specific orderings and tests on our Pet-vectors.

Lambda Specification. An important aspect of this lab is the specification of small bits of code that determine, for example, how we compare two Pets. These “bits” are really just functions that are specified by methods of a dedicated “wrapper” object. When we want to specify the way to compare two items (say Pets), we use a `Comparator<Pet>` object’s `compare(Pet a, Pet b)` method. When we want to provide a method for testing a Pet condition, we specify a `test(Pet p)` method in a `Predicate<Pet>` object.

The `SortingVector<E>`'s sort method takes a `Comparator<E>`:

```java
public static sort(Comparator<E> c)
```

When we call this method we must specify `c`. Suppose we have a `SortingVector<String>` and want to build a `Comparator<String>` that compares String objects based on length. We could use this definition, stored in a file called `StrLenComp.java`:

```java
import java.util.Comparator;
public class StrLenComp extends Comparator<String> {
    public int compare(String a, String b) {
        return a.length()-b.length();
    }
}
\(^1\)https://www.kaggle.com/aaronschlegel/seattle-pet-licenses
\(^2\)java.util.Comparator
\(^3\)java.util.function.Predicate
The `compare(a,b)` method (like the `a.compareTo(b)` method for Strings and similar classes that implement the `Comparable` interface) returns a value that is less-than, equal-to, or greater-than zero if `a` is, respectively, less-than, equal-to, or greater-than `b`. If you think about this a bit, the difference between `a` and `b` does just this.

Now, in calling a sort that would make use of this definition, we would use the code

```java
tuple.sort(new StrLenComp());
```

This "new" object is simply a mechanism for delivering the `compare(a,b)` method to the `sort` routine. This is pretty wild: we're treating a bit of code—to be used later—as though it were data. This is our very first step along the road to functional programming.

Beginning in Java 8, when we reference an object from a class that has one abstract method declared (in our example, here, `java.util.Comparator<E>`, has a unique missing method, `compare(E a,E b)`), we can provide the missing code in a compressed description called a lambda specification. Here's an anonymous one-time lambda definition that is equivalent to the `StrLenComp` class definition we gave above:

```java
tuple.sort( (String a, String b) -> {
    int al = a.length();
    int bl = b.length();
    return al-bl;
  } );
```

This syntax is specifying the body of the routine with the signature specified—two `String` objects, `a` and `b`. The arrow (`->`) is used to signal a lambda specification and is followed by a basic block of Java code with a return. Since the `compare` method must return an `int`, the value `al-bl` is appropriate.

It is hard to overstate what is happening here: at the point where a `Comparator<String>` is needed, we pass sort a snippet of code as the actual parameter. How is the code converted to a `Comparator` instance? The compiler writes an anonymous extension of the `Comparator<String>` class, with the indicated code as the body of the unique abstract method, and an instance of that class is constructed and passed to the sort method. Once the sort method is finished, the class definition and its object are recycled. Wow.

When the basic block can be written as a single return statement we can simplify it further: just specify the value returned:

```java
tuple.sort( (String a, String b) -> ( a.length()-b.length() ) );
```

The parentheses around the return value are unnecessary, but we find it helps with readability.

In this lab, we'll make use of one other abstract class, the `Predicate<E>`\(^4\). The purpose of this class is to support the specification of functions that test a condition of the type `E`. If you browse the interface for this class you will see one abstract method:

\(^4\)If you've used Python, you've probably learned about lambda syntax for specifying one-off functions. This is Java's tardy but concise attempt at the same thing. The use of the term `lambda` is an homage to Alonzo Church's important formulation of the Lambda Calculus, a mathematical study of function-based computation long before we had physical computers.

\(^5\)The purpose of this class is to support the specification of functions that test a condition of the type `E`. If you browse the interface for this class you will see one abstract method:
public abstract boolean test(E);

The test(E value) method takes a single value and returns true or false depending on the code ultimately provided in the lambda specification. We'll use this to help us specify how to select or filter objects that will form a new SortingVector. Obviously, there are lots of ways to select elements from a vector—depending on the application—so specifying a last-minute or one-off lambda expression is ideal for this purpose.

This Week's Tasks. Here is what we expect you to do, this week:

1. Clone your lab4 repository:

   cd ~/cs136
git clone https://evolene.cs.williams.edu/cs136-labs/22xyz3/lab4.git

   replacing 22xyz3 with your CS username. This will create a subdirectory, lab4 in your cs136 directory. Looking around this directory, you'll see the following files:

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SortingVector.java</td>
<td>the basis for your sort-augmented Vector</td>
</tr>
<tr>
<td>InsertionSort.java</td>
<td>an example of insertion sort</td>
</tr>
<tr>
<td>SeattlePets.csv</td>
<td>the database of Seattle pet registrations, in CSV form</td>
</tr>
<tr>
<td>CSVReader.java</td>
<td>code we have written to read CSV files</td>
</tr>
<tr>
<td>Pet.java</td>
<td>the class that produces a Pet object</td>
</tr>
<tr>
<td>qi.java</td>
<td>starter code for question-answering applications.</td>
</tr>
</tbody>
</table>

Browse through these files; there's a lot to see here. The main portion of the lab will the completion of the SortingVector and Pet classes.

2. Let's start with SortingVector.java. Add a sort(Comparator<E> c) method to the SortingVector<E> class. This public method should sort the elements of the Vector<E> this class extends. We have put the code for an array-based insertion sort in the file InsertionSort.java. We make two important observations:

   (a) The insertion sort we provide sorts a user-provided array of integers. Your version should sort the internal elements of this object. We're part of a Vector, not processing a user's array.

   (b) The insertion sort code performs direct comparisons of data. So, for example, if you see

      if (temp < data[index-1]) ...

      that code should be converted to a call to the Comparator c's compare method. As mentioned before, comparison methods, like compare(a,b) in Java typically return a value that is less-than, equal-to, or greater-than zero if a is, respectively, less-than, equal-to, or greater-than b. Thus we might end up with

      if (c.compare(temp,get(index-1)) < 0) ...

---

Insertion sort is sufficient for our purposes here, but it can be slow. Suggestion: be patient. You can use any sorting technique you wish, as long as it is stable: a stable sort does not allow equal values to “pass each other” during the sorting process. Bubble sort and insertion sort are stable. Selection sort and quicksort are not (the way we've seen them). Mergesort, with care, can be. See the example program stable.java and Section 6.7 in the textbook for more details.
Notice the parallel between the direct array element comparison and the equivalent compare-based comparison with zero. If you do not understand this, seek help!

When you’ve completed the sort method, observe that the main method of SortingVector sorts a vector of 20 random integers:

```java
values.sort((Integer a, Integer b)->(a-b)); // Comparator lambda
```

Notice how the compiler automatically converts the Integer expression a-b to the required int value. If you run the SortedVector’s main method:

```java
javac SortedVector.java
java SortedVector
```

it will print, in ascending order, 20 random non-negative integers less than 1000.

3. Write the filter(Predicate<E> p) method for SortedVector<E>. This public method returns a new SortedVector<E> whose elements are identified by the java.util.function.Predicate<E> p. Assuming that p is a fully specified Predicate, the body of your method should run through all the elements of this SortedVector and copy those identified by the p to the result vector.

The code we provided does none of this. It simply returns this, ignoring the predicate p.

When your filter method is finished, the lines in main:

```java
SortingVector<Integer> mults;
mults = values.filter( (Integer a) -> (a%3 == 0) );
```

assign mults to be a new SortingVector<Integer> whose ascending random values are divisible by 3. The selection of values to construct mults is specified by the Predicate lambda specification. We are thankful that the compiler is helping us with this concise specification. Again, if you do not understand what is going on here, ask questions.

Notice that every time you run SortingVector the program potentially generates different numbers of random multiples of 3.

4. It is worthwhile stopping and thinking about the versatility of this new extended class. Even when the values stored in the SortingVector have no obvious ordering (we say the values are not totally ordered), we can specify code that can be used to order the values in an application-specific way.

In fact, you could sort the values using several different keys. Suppose we had a SortedVector of String objects and performed the following two sorts:

```java
v.sort( (String a, String b) -> (a.compareTo(b)) ); // natural order
v.sort( (String a, String b) -> (a.length()-b.length())); // length order
```

You can find this code in `stable.java`. Compile, run, and type text into the program. Indicated end of input with control-D.
we would find the strings are ordered primarily by length. Closer inspection of each section of Strings that are the same length, we notice they're in alphabetical order! As mentioned above, this is a feature of stable sorts. With care, we can use many different lambdas to sort collections of values in many different orders.

5. Let's think about pets. Examine the SeattlePets.csv file. The lines of this file have seven fields separated by commas:

```
December 18 2015,S107948,Zen,Cat,Domestic Longhair,Mix,98117
```

Here's how we interpret these fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Date of pet registration (e.g. &quot;July 24 2019&quot;)</td>
</tr>
<tr>
<td>1</td>
<td>Registration ID (e.g. &quot;S107948&quot;)</td>
</tr>
<tr>
<td>2</td>
<td>Pet Name (e.g. &quot;Java&quot;)</td>
</tr>
<tr>
<td>3</td>
<td>Pet Species (e.g. &quot;Cat&quot;)</td>
</tr>
<tr>
<td>4</td>
<td>Pet Breed (e.g. &quot;Domestic Shorthair&quot;)</td>
</tr>
<tr>
<td>5</td>
<td>Secondary Breed Description (e.g. &quot;Mix&quot;)</td>
</tr>
<tr>
<td>6</td>
<td>Pet Zipcode, if known (e.g. &quot;98119&quot;)</td>
</tr>
</tbody>
</table>

We hope to be able to read in the SeattlePets.csv database using a CSVReader. The CSVReader reads each line from a CSV file and returns a Vector<String> of the strings that make up each field. Here's a typical use of the CSVReader class:

```java
try {
    Scanner input = new Scanner(new File("SeattlePets.csv"),"UTF-8");
    CSVReader r = new CSVReader(input);
    for (Vector<String> line : r) {
        ... process a pet ...
    }
} catch (FileNotFoundException e) {
    Assert.fail(e.toString());
}
```

The Scanner (input) allows us to scan through a file, looking for words or tokens. The CSVReader interprets the tokens on each line as a description of a Vector<String> of fields that are separated by commas. The try-catch statement is used to catch errors that occur if "SeattlePets.csv" is not found in the current directory. Fortunately, we've provided the code that manages the CSV reader for our Pet project, but you may find code similar to this useful in other classes.

Our primary focus will be on developing a Pet object that will allow us to maintain a database of pets from Seattle. The next few steps take us through that process.

6. Peruse Pet.java. Here, we see the start of a class, Pet, that gathers the information from each line of the Seattle pet registration database. Notice that it compiles even though there are no instance variables associated with the Pet object. When you run the program, it is obviously trying to print our favorite pet, found at location 25, 348 in the vector of pets read from the database:
Anticipation builds.

7. The Pet constructor is called from the static readPets method. This constructor takes the pet-describing Vector<String> that is returned by a CSVReader scanning through the SeattlePets.csv database. This Vector has seven fields described in the comment for the constructor. Some fields should be saved in instance variables as String objects, field 5 should be ignored, and one—the ZIP code—is to be maintained as an int with a missing value of zero.

Declare the half-dozen protected instance variables and initialize them in the constructor.

8. Each field has a public accessor method. For example, name() returns the pet’s name, as defined by field 2 in CSV Vector. Complete the accessor methods.

If all works well, you should be able to compile and run the Pet.java main method to print pet 25,348, our current favorite:

```
javac Pet.java  
java Pet
```

9. Notice that, in main, we make use of the static readPets method to read the database as a Vector<Pet>. In fact, the result is a SortingVector<Pet>. We should be able to leverage this fact to call sort and filter methods to sort the database and focus on pets of interest.

We would like you to write standalone applications that answer the following four questions. Your programs should be very simple; we believe you do not need to use any loops:

q1. In q1.java, write a program that prints out pets who live at the University of Washington, with ZIP code 98195. Notice they’re not huskies.

q2. In q2.java, write a program that prints out the unique pet with the longest name.

q3. In q3.java, write a program that prints out the pet named “Java” whose ZIP code is largest.

q4. Many pets in Seattle are named Luna. Are more Lunas cats or dogs? Write a program q4.java that answers this question.

If you make it this far, you’ll have earned 9 points, most of the credit for this lab! Stop here, or continue on for full credit and a couple of extra challenge questions to answer.

10. Sometimes it is useful to build a filter that finds unique values. If we sort values using a Comparator, we will potentially see regions of adjacent values, a and b, that cannot be distinguished by the Comparator: c.compare(a,b) returns zero because these values appear to be equal. At the boundaries of these regions, however, we see adjacent values that can be distinguished by the Comparator.

Implement the unique(Comparator c) method that returns a SortingVector<E> of one or more values that are representative of the regions values that appear in the sorted list. For example, the following code
SortingVector<Integer> data = new SortingVector<Integer>();
int[] nums = {3, 2, 3, 45, 5, 4, 3, 2, 4, 4};
for (Integer i : nums) data.add(i);
SortingVector<Integer> reps =
    data.unique((Integer a, Integer b) -> (a-b));
for (Integer r : reps) System.out.print(r+" ");
System.out.println();
leaves data unchanged, but prints reps as
2 3 4 5 45
the list of values that are pairwise-distinct using the Comparator. Be careful to make sure that your
approach works even when the SortingVector has a single value or is empty.

11. To test your implementation of unique, answer the following questions:
   
q5. Write a program q5.java that prints a representative of each of the pet species registered in
   Seattle. There are four.

q6. Write a program q6.java that prints an example of each of the seven Seattle pet breeds that
   contain the string "Retriever".

If you've made it this far: Congratulations! Make sure you add, commit, and push SortingVector.java
and programs q5.java and q6.java.

Submitting Your Work. Make sure you add and commit your SortingVector.java, Pet.java, and
your question-answering programs q1.java, q2.java, q3.java, and q4.java. If you answered questions
5 and/or 6, include q5.java and q6.java as well. In any case, sign the honorcode.txt file. Remember
to push your work each time you leave the lab.