In this laboratory, we will use our Stopwatch class to measure the efficiency of the Vector class. Specifically, we want to see how execution speed is affected by the increment. Recall that increment is the amount by which the underlying data array is lengthened when the vector requires more space. If increment is set to zero, then the size of the data array is doubled. We’ll be using the Vector class since the ArrayList class only doubles the array length and does not give you incremental building as an option.

Begin by closing all of your open Eclipse projects, so that errors in them will not affect your work today. Use Projects/Close.

Create a new Eclipse project named something like Lab2. Remember to continue to the window in which you can add the BAILEY variable. Next, copy the file /common/cs/cs062/labs/lab2/StopWatch.java into the src directory in your new Eclipse project and select File/Refresh.

Start a new class VectorTimer, which will be simply a vessel for the main method and a few other static methods:

- **public static long run(int maxSize, int increment)**
  The run method creates a new empty vector of type Vector<String> with the specified increment. It returns the time that it takes to add maxSize strings to the Vector. Use the Vector<String> method add, and always add the same constant string—your name, for example. To attempt to minimize the impact from garbage collection add the line: System.gc(); in your run method right before you start the timer.

- **public static ArrayList<Long> trial(int size, ArrayList<Integer> incrs)**
The trial method compares the results from run for a fixed size and varying increments. It makes one call to run for each entry in the incrs vector. The results are returned in an ArrayList whose size is the same as that of incrs.

- public static void main(String[] args)

  The main method runs several trials and prints the results. Start with increments of 1, 10, and 0; and sizes of 0, 5000, 10000, 15000, . . . . You may want to adjust the sizes when you see the results.

  Present the output in a table like the one below; see the note below about formatting. The nanosecond precision of Stopwatch is too fine; you will need to adjust the scale of the timing values as they are printed, which can vary computer to computer.

<table>
<thead>
<tr>
<th>size</th>
<th>linear (1)</th>
<th>linear (10)</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5000</td>
<td>148</td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>10000</td>
<td>580</td>
<td>58</td>
<td>0</td>
</tr>
<tr>
<td>15000</td>
<td>1321</td>
<td>132</td>
<td>0</td>
</tr>
<tr>
<td>20000</td>
<td>2733</td>
<td>267</td>
<td>1</td>
</tr>
<tr>
<td>25000</td>
<td>4863</td>
<td>491</td>
<td>1</td>
</tr>
<tr>
<td>30000</td>
<td>7781</td>
<td>781</td>
<td>1</td>
</tr>
</tbody>
</table>

  We will discuss the significance of your results, and those of your classmates, as they appear. Some things to think about: what is the running time (i.e. Big-O running time) of increment vs. double? Does your data accurately reflect this?

More fun...

Once you’ve got all this working, if you have time we can try out a few additional things:

- What happens with other increments (besides 1 and 10)? Can you predict what the results will look like, for example what do you think a column headed linear (100) would look like?

- Rather than just running one experiment per setting, you can run multiple experiments (say 5 or 10) and average the results in your run method. This will be a bit slower, but should give you more accurate results.
It may be interesting to compare the performance difference between ArrayList and Vector. ArrayList does NOT allow you to adjust the increment size; it always doubles the size. However, you can compare the performance of Vector vs. ArrayList for doubling sizes. Which is faster?

A note on formatting textual output. The object System.out has type PrintStream, which in turn has a method format. format is very general and makes it easy to print the lines in the table. The call

    System.out.format("First: %8d, second: %-12s%n", num, str);

creates a string and prints it. The string is formed by

- replacing %8d with the numerical value of num, right justified in a field eight characters wide, and
- replacing %-12s with the string representation of str, left justified in a field twelve characters wide.

If num and str are 47 and XLVII respectively, then

    First: 47, second: XLVII

is the result of the method call above.

The letters after the percent sign, d and s in this example, indicate the kind of data being formatted; they are not variables. The sequence %n is the OS independent newline character. You may have as many % expressions in the format string as you want; they are matched with the arguments that follow. There are many more options for format strings; see the Java documentation for the classes PrintStream and Formatter or the tutorial at http://java.sun.com/docs/books/tutorial/java/data/numberformat.html for more information.