CMSC 132 Homework #1 Fall 2012

Date due: Thursday, September 13, 10:00:00 p.m.

1 Introduction and purpose

In this homework you will write some methods that use features of Java that were covered in CMSC 131, and just a few concepts that have been covered so far this semester. You will also become familiar, if you aren't already, with the mechanism used to access or check out the starting code for an assignment from your CVS repository into Eclipse, to use Eclipse to code in Java, and to submit projects to the CMSC project submission server. Lastly you will write tests of your own that exercise your methods in ways that the provided public tests discussed below do not. Hopefully you will not find the methods that hard to write. The homework's purpose is just to get started writing code for this course.

This description is rather long, because the first programming assignment needs to explain a number of details for those who may not be familiar with how to do things like accessing an assignment's starting code from a CVS repository, writing and running JUnit tests, etc. Notice that the actual description of the methods you have to write is just two pages of this assignment. There is also a separate document, that will be provided soon, describing how programming assignments should be written and will be graded, which you should read **carefully**.

Due to time constraints and other factors it is not practical to provide detailed information or assistance regarding programming assignments via email, so please ask questions you may have about this homework assignment in person, either during office hours, or before or after class.

2 Checking out the homework starting code

First you need to retrieve the starting code for this homework, which contains the tests (public tests) your homework will be run on, as well as skeletons of the classes that you have to write. These steps assume you have already set up your CVS repository in Eclipse, as described in the handout "CVS repository setup for CMSC 132" available under "Course Documents" on ELMS. If you haven't, do those steps first.

1. Start Eclipse and change to the CVS perspective by selecting in Eclipse **Window** → **Open Perspective** → **Other** → **CVS Repository Exploring**. You should see the CVS repository location that you created by following the steps in the handout referred to above.
2. Click on the little triangle at the start of the line identifying the repository (depending on your operating system, it might be a little '+' rather than a triangle). You should see at least three lines with the names "HEAD", "BRANCHES", and "VERSIONS".
3. Click on the little triangle (or '+') at the start of the line that says "HEAD".
4. You may see "Pending..." as data is loaded from the repository (you'll have to enter the secure storage password if you needed to create one manually when setting up your CVS repository), then you'll see at least two folders.
5. Right–click on the folder for the homework or project that you want to check out (`hw1` for this homework) and select **Check Out**.
6. After this you should have a new project in your workspace that has the same name as what you checked out (`hw1` for this homework). You can change to the Java perspective to start working on it by choosing **Window** → **Open Perspective** → **Other** → **Java (default)**, then clicking on that project.

Note that you only need to use the CVS perspective once for each assignment to be written, although if you use different computers you can check out an assignment on each one you use, by connecting to your CVS repository and following the steps above.

As soon as you check out the homework starting code from your CVS repository try to run the public tests, as described below in Section 5. Of course they won't produce the right results, but at least this should let you know if you have some type of problem with your Eclipse setup so you can get it resolved immediately, rather than it occurring just before the homework is due.

3 Brief description of problem domain

This homework involves writing some methods to construct and manipulate a simple index of words. The problem isn't all that interesting, but it does involve using basic concepts from CMSC 131, such as objects that contain and use other objects, as well as the `ArrayList` class. An index is a listing of all of the words in a document, with some type of list stored for each word of the numbers of all the pages that that word appeared on. For simplicity, assume the homework will not involve duplicate page numbers for any word (no word will have the same page number associated with it twice). For example, suppose a short document with very small pages contained the following text (apologies to Charles Dickens for slightly modifying his work):

It was the best of times, it was the worst of times; it was the age of wisdom, it was the age of foolishness.	It was the epoch of belief, it was the epoch of incredulity; it was the season of Light, it was the season of Darkness.	It was the spring of hope, it was the winter of despair; we had everything before us, we had nothing before us; we were all going directly to Heaven, we were all going the other way.
page 1	page 2	page 3

Then an index of this work could conceptually be something like the following:

age:	1	had:	3	the:	1 2 3
all:	3	Heaven:	3	times:	1
before:	3	hope:	3	to:	3
belief:	2	incredulity:	2	us:	3
best:	1	It:	1 2 3	was:	1 2 3
Darkness:	2	it:	1 2 3	way:	3
despair:	3	Light:	2	we:	3
directly:	3	nothing:	3	were:	3
epoch:	2	of:	1 2 3	winter:	3
everything:	3	other:	3	wisdom:	1
foolishness:	1	season:	2	worst:	1
going:	3	spring:	3		

The `hw1` project contains an `Index` class that you will use to implement an index. An `Index` just contains an `ArrayList` of something called a `WordEntry`, which is a class we didn't write that you need to create (there's just a skeleton for it in the project); each `WordEntry` object will need to contain one word plus the list of page numbers on which that word appeared. You can add whatever fields and write whatever methods you need in the `WordEntry` class– given the constraints in Section 8 below– that will allow you to write the methods in the `Index` class.

Rather than performing input in Java to read lines of text from a document, and break them up into their consitutent words, we will just simulate building an `Index` by passing appropriate arguments into a method `addWord()`; its arguments will be a word and an array of `int`s representing the numbers of the pages that that word appeared on in a document. This is the only method that adds something to an `Index`. There is also a method that modifies an `Index` by removing a word, but other than these two methods the remaining ones in the `Index` class just report or return information about an `Index`, rather than altering it.

4 Description of methods to be written

Each method you have to write in the `Index` class initially just throws an `UnsupportedOperationException`. Remove the exception, and write the body of each method. All the methods you have to write are public.

The methods that have objects or arrays as parameters must throw an `InvalidParameterException`, without changing anything in its current object or its parameters, if any object or array parameter is `null`.

This is a checked exception that we defined, which you should not modify (it is in fact empty), that will be used to signal that a method's parameters are invalid. It doesn't matter what string or message the exception is constructed with, or even if it has a message string at all.

4.1 `Index addWord(String word, int[] pages)`

As mentioned above this method will add a word to its current object `Index`. The array `pages` represents the list of pages on which the word appears. The method should construct a new `WordEntry` and add it to the end of the `ArrayList` field `wordList` that stores an `Index`'s list of words; therefore the parameter `word` becomes the index's new last word. You may assume for simplicity that `word` is a string without any punctuation or whitespace characters. The new word's list of page numbers is to be stored in exactly the same order that the elements of `pages` appear in. You may assume, without having to check, that the values in the array `pages` will be in nondecreasing numerical order.

If the word `word` is already present in the method's current object, it should **not** be added again, and the method should just have **no effect** (even if the values in `pages` are different from the pages that are already being stored for the word). If `pages` contains any values that appear more than once, only **one occurrence** should be added. For example, if the word "banana" is added with an array of page numbers containing "1 3 4 4 5 6 6 6", the values that would actually be stored in the new `WordEntry` would just be "1 3 4 5 6". Note that as a result of the above, no duplicate words or word occurrences can ever be present in an `Index`, so even if a word appeared many times on the same page of a document, that page number would only be stored once for that word in the index.

The method should return a **reference to its current object**, to allow chained method calls. As mentioned above, all methods should throw an `InvalidParameterException` if any parameter (either `word` or `pages` for this method) is `null`. This method should also throw an `InvalidParameterException` if `word` is an empty string, or if `pages` is a non–`null` array but one that has zero elements (which is possible in Java).

This method will not check whether `word` is being added in position such that the words in the index will be in alphabetical order, or whether the values in `pages` are in any particular order. (It will only check whether `word` is already present, or `pages` has duplicate elements, and avoid adding multiple instances if so.) If the user of your class wants their index to have these properties it will be up to them to make calls to your method in the right order and with the right parameters to ensure it. Note though that there is a method to be implemented, discussed below, that will test whether the alphabetical property is true for an `Index`.

4.2 `String longestWord()`

This method should return (a reference to) the longest word in its current object `Index`, meaning the one with the most characters. If there is more than one word that has the same (longest) length, the one appearing **earliest** in `wordList` (which will be the one that was added to the index first) is the one that should be returned. If the index does not contain any words at all the method should just return an empty string.

4.3 `int numUniqueWords()`

This method should return the number of **unique** words that are stored in its current object `Index`; the result will be zero or more. By "number of unique words" we mean the count of the number of words, not taking into account that some words migh occur on more than one page. For the example above on page 2 the method would return 35.

4.4 `int numWordOccurrences()`

This method should return the total number of word occurrences in its current object index, where each page that a word appears on will be counted once; the result will be zero or more. In other words, this method will count **every** page that a word is on if it appears on multiple pages of the document that the index is for. For the example on page 2 above the method would return 45.

4.5 `boolean isAlphabetized()`

This method should return false if any word in its current object `Index` is out of alphabetical order, and true otherwise. For our purposes alphabetical order means that every word is strictly greater than the ones preceding it in `wordList` when they are compared using the `String` class method `compareToIgnoreCase()`, which ignores differences between uppercase and lowercase. (Using the `compareTo()` method in the `String` class would require that all the words beginning with uppercase letters appear first in an index for it to be considered to be in alphabetical order, because that is the order the characters appear in the ASCII character set, but it's not usually how words are listed in a book's index.) This method should actually return false for the example on page 2, because it contains both the words "It" and "it". If an index has no words at all, then no words can be out of alphabetical order, so true should be returned.

4.6 `void removeWord(String word)`

This method will remove a word and all its associated page numbers from its current object `Index`. The number of words reported by `numUniqueWords()` would subsequently decrease by 1. If `word` is not present in the index that the method is called upon, it should just have no effect. If the last or only word is removed from an index then it would subsequently contain no words at all. Note that case is significant, so `word` must be exactly identical to a word in the index for it to be removed; for instance, if `removeWord("darkness")` was called on the example index on page 2 the call would have no effect.

4.7 `String toString()`

The `toString()` method overrides the `toString()` method from the `Object` class, and should return a string representation of the current object `Index`. The string should be constructed as follows:

- Each word in the index should appear in the returned string, in the same order that they appear in `wordList`.
- Each word must be followed immediately by a colon, which is followed by a single blank space and the list of page numbers associated with that word (there will be one or more), in the same order that they are stored in the **WordEntry** for that word (which is the same order that they appeared in the array `pages` that was passed into `addWord()` when that word was added).
- A single blank space must separate each pair of page numbers, but no blank space should follow the last page number. (**Note**: this method **should not** add a blank space after every number and then remove the last one; it should instead just not add a blank space after the last number in the string.)
- The data for every word (the word, colon, and blank–space–separated list of page numbers for that word) in the string being returned should be terminated by a newline character. Therefore if an index contains n words ($n \geq 0$) the string returned by this method should contain n newline characters.

The above implies that if an index contains no words at all the string returned by this method should be an empty string (without a newline).

For example, if an index is storing the word "banana" with the list of pages "1 3 5 7 9", followed by the word "cherry" with the list of words "3 4 6", the string returned by `toString` should be (blank spaces are shown as ␣) "`banana:␣1␣3␣5␣7␣9\ncherry:3␣4␣6\n`" (the quotes are just to show what the string consists of, and are not part of the string to be returned).

5 Running your code, and homework tests

If you navigate to the `tests` package of the Java project created for this homework you'll find the Java source file `PublicTests.java`, which contains the public tests for this homework. Programming coursework in this class may be run on two types of tests, which are called public tests and secret tests. (Unlike in CMSC 131, this course will not use any release tests.) Public tests are test cases that are provided to you when a programming assignment is distributed, so you can run your program on them and verify before the assignment's deadline

whether it produces the right results; of course if it does not you can correct any problematic cases. Secret tests are additional test cases that your program will be run on, but you will not see them or their results until after the assignment's due date has passed. Obviously the better you test your code yourself before submitting an assignment, the more likely it is to get correct results on the secret tests.

The public tests for most programming coursework will be in the form of JUnit tests. The next section explains how to write your own JUnit tests. If you haven't used JUnit before, or haven't used it with Eclipse, here's how to run your program on JUnit tests like the public tests:

- In Eclipse, open the package `tests` for the homework and either:
  - left–click on the `PublicTests.java` source file (to highlight it), then right–click and choose **Run As → JUnit Test**, or
  - open the `PublicTests.java` source file and, while it is highlighted (click on its tab), use the **Run** menu at the top of Eclipse to choose **Run As → JUnit Test**.
- If your program contains any syntax errors Eclipse will pop up a window that says that errors exist in the project; you can select **Proceed** to see the compilation errors, or look for the little X's in red circles in the left column of your code and hover the mouse over them to see a description of the problems.
- Assuming your program compiles, if it produces the right results for all the JUnit tests you will see a green bar in the left JUnit pane, and under that you will see **Runs** with two numbers that are the same, for example 10/10 means your code produced correct runs for 10 tests out of the 10 that were run. Next to that, the number of errors and failures will both be zero.
- If your program had problems running any of the tests, either the number of errors or the number of failures, or both, will be greater than zero, and you will see a line in the JUnit pane for each test. Tests that your program worked correctly for will have a small green check next to their name, those for which it had an error will have a red X, and those for which it encountered a failure will have a blue X.

  An error means that your code produced some type of Java error while being run on the tests, such as a `null` reference exception. You can left–click on the name of that test in the JUnit pane to see a description of the error, as well as the line of your code where it appeared, at the bottom of the JUnit pane (labeled **Failure Trace**). Double–click on one of the lines of code to jump there. Obviously you'll need to figure out why your program had the error and try to correct it.

  A failure means your code correctly ran to completion but failed some JUnit assertion or property being checked while being run on the tests. Just like an error, you can click on the name of the failure to see a description of what property wasn't satisfied and where it was checked, so you can fix your code.

6 Submitting programming coursework

Even after you (hopefully) get your program to work correctly on all of the public tests, and write tests of your own as described in the next section, we won't have a copy of it– meaning you won't receive a grade for it– until you submit it. To submit a programming assignment, assuming you've installed the necessary Eclipse plugin (or installed the Eclipse version for Windows with the plugin already present) then you can submit your assignment by switching to the Java perspective, right–clicking on your project folder in the package explorer, and selecting **Submit Project...**. Note that the first time you submit an assignment you must enter your UMCP directory ID and password, but these are saved and not needed for subsequent submissions of that same assignment. **Note:** if the **Submit project** menu item doesn't appear then you must not have the Course Management plugin installed; see the CMSC department's Eclipse installation instructions also available under "Course Documents" on ELMS.

Note: <u>do not</u> use the submit server's web submission procedure to submit a jarfile or zip archive, since problems may occur with assignments submitted this way. If you have trouble submitting assignments using the procedure above you must get the problem resolved by coming to office hours, rather than submitting a jarfile or zip archive. If you submit a jarfile or zip archive do not expect to receive any credit for your student tests (mentioned below), regardless of whether they gave the right results when you ran them in Eclipse.

Once you submit, you **must** log in to the CMSC department project submission server, reachable by clicking on https://submit.cs.umd.edu. The submission server runs your submission on the public tests, and allows you to see the results. We can only grade what's submitted to us, based on how it works on the project submission server, so you **must** be sure to log into it right after you submit, to check your assignment's results there. Although your code should work correctly on the submit server if it works correctly in Eclipse, there are certain types of errors that could make your program work right for you, but not work when it's submitted, so be certain to log into the submit server to verify that everything is OK there.

7 Writing JUnit tests

The following material is taken from the CMSC department's Eclipse tutorial, at:

www.cs.umd.edu/eclipse/EclipseTutorial/JUnitTesting.pdf

with minor revisions.

Besides running your code on the public tests provided to you, you should write your own tests ("student tests") to check its behavior in situations that the public tests don't test for. Your tests should be JUnit tests like the public tests. You will be receiving credit for your tests, so if you don't write any you will lose all of those points.

So far you have probably run JUnit tests (either in CMSC 131, or just running your code on the public tests for this homework); here we describe how to write JUnit tests as well. Of course the most important part is thinking about the methods to be tested, and identifying as many behaviors to be tested as you can. When you have, here's how to test those behaviors using JUnit.

We already created an empty class `StudentTests.java` in the `tests` package of this homework. You should just write your student tests in it. If you have to create your own student tests for later programming coursework you would have to add a JUnit Java source code (`.java`) test file to the Eclipse project as follows:

- From the **File** menu, choose **New** and **JUnit Test Case**. Be sure that "New JUnit 4 test" is selected (rather than JUnit 3).
- To facilitate grading, **always use the name `StudentTests.java` for your student tests** for any homeworks or projects in this course, so enter `StudentTests` for the test class' name. Due to the way the submit server works you may not receive credit for student tests unless you use this **exact** name, spelled and capitalized **exactly** as shown.
- Click on **Finish**. If a popup appears telling you that the JUnit library is not on the build path, select **Perform the following action** for **Add JUnit 4 library to the build path** and click on **OK**.
- At this point you will see a class declaration that should look as follows:

```
import static org.junit.Assert.*;

public class StudentTests {

  @Test
  public void test() {
    fail("Not yet implemented");
  }

}
```

Then to actually write JUnit tests in the your `StudentTests` class:

- To create tests define public void methods in your `StudentTests` class whose names begin with the word "test" (e.g., `testLongestWord`, `testStudent1`, `testNumberTwo`, etc.). Each of these methods will be a test. The method `test()` shown above is just a dummy automatically–generated test, which you can remove when you write your own.

- Inside a test method you can have typical Java code, where assertions are used to specify the expected results from the test.
- The following are common assertions used in JUnit tests:

  `assertTrue:` verifies that the argument expression is true. If the argument expression is false the test fails, otherwise execution is successful.

  `assertNull:` verifies that the argument expression is `null`. If the argument expression is not `null` the test fails, otherwise execution is successful.

  `assertNotNull:` verifies that the argument expression is not `null`. If the argument expression is `null` the test fails, otherwise execution is successful.

  `assertEquals:` takes two arguments (the expected value and the actual value). If the values are not equal the test fails, otherwise execution is successful.

  In Eclipse, type `this.assert` (and then wait) and you will see a list of the assert methods available.
- You can define auxiliary methods (e.g., private methods) that support the set of tests you are developing.
- You can have multiple assertions in a JUnit test. Once the first one fails the whole test is considered to had failed.
- Note that the last public test for this homework illustrates how, in JUnit 4, a test can specify that it should pass only if a certain exception is thrown.
- Beware– static data persists across JUnit tests, so be very careful about using any type of static data in your `StudentTests` class. (This will probably cause one test's results to interfere with other test's.)

Here's an example of a small class `AuxMath`, followed by a `JUnit` test example class `StudentTests`:

```
public class AuxMath {

  public static int maximum(int x, int y) {
    return x > y ? x : y;
  }

  public static int minimum(int x, int y) {
    return x > y ? y : x;
  }

}
```

```
import static org.junit.Assert.*;

public class StudentTests {

  @Test public void testOneMaximum() {
    assertEquals(20, AuxMath.maximum(10, 20));
  }

  @Test public void testTwoMinimum() {
    assertEquals(5, AuxMath.minimum(30, 5));
  }

}
```

8 Homework requirements

1. To keep things simple you should only use the features of Java covered so far this semester and in CMSC 131. You may use Java arrays, and the `ArrayList` class, but **do not** use any other Java collections.

2. You **must** use the `ArrayList` field `wordList` to store the list of words in an index, and you **may not** change its definition, so you **must** create the class `wordEntry` that `wordList` uses.

3. Do **not** add any other classes or packages, since things sometimes don't work right on the submit server if the files and packages submitted are not what it is expecting. You may lose significant credit for your homework if you add other classes or packages.

4. You may add any **private** fields and methods to the `Index` class, and any public, private, or package methods to the `WordEntry` class that you have to write, but do not add any public fields or methods to the `Index` class, or any public fields to the `WordEntry` class.

5. For this homework you will not be graded on programming style. Your grade will be based on:

public tests	60 points
secret tests	30 points
your own student tests	10 points

6. For this homework there is no minimum number of student tests, and no required level of code coverage they should achieve. However, you should attempt to be as thorough as you can in writing student tests. Ideally they would test every method in every case that you can identify that the public tests do not already test. Student tests that exhibit a **reasonable attempt** to test your code well should receive full credit.

7. If you have a bug in your code that you can't figure out, and need to come to office hours, you **must** have written **at least one student test** that illustrates the problem, otherwise we will not be able to assist you until you have written one or more such tests.

   If you have a bug requiring office hours assistance you must also be prepared to show the TAs how you tried debugging your code using Eclipse's debugger, and what results you found in the process.

8. The course project grading policy will be provided shortly on ELMS, describing how programming assignments will be graded, late penalties, which version is graded if you submit more than once, etc. Be sure to carefully read this as soon as it's available.

9. During coding you are encouraged to **frequently** save your work to your CVS repository as described in the last section of the handout about setting up your CVS repository. The Course Management Plugin is supposed to save copies of your work to your CVS repository every time you run your code, but for reasons that are uncertain it doesn't always do so, and it should be saved more frequently in any event.

10. **Do not wait** until the last minute to finish and submit your homework! It is strongly suggested that you finish and submit it **at least** one day before the deadline, to allow time to reread the assignment and ensure you have not missed anything that could cause you to lose credit. If you have, this will give you time to correct it. This will also give you time to get any last–minute unexpected problems fixed.

11. **Do not make unnecessary submissions** of the homework. See the project grading policy to be posted for full details. You may **lose credit** if you submit this homework an excessive number of times.

9 Academic integrity

You are allowed to work together or to provide or receive help from anyone else in installing Eclipse, setting up your CVS repository, and checking out the starting code for this homework. But everything else– namely

your implementations of the method to be written, **as well as** all student–written tests– must be strictly **your individual work only**, other than possible assistance from the instructional staff during office hours.

Please **carefully read** the academic honesty section of the course syllabus. **Any evidence** of impermissible cooperation on projects, use of disallowed materials or resources, or unauthorized use of computer accounts, **will be submitted** to the Student Honor Council, which could result in an XF for the course, or suspension or expulsion from the University. Be sure you understand what you are and what you are not permitted to do in regards to academic integrity when it comes to project assignments. These policies apply to all students, and the Student Honor Council does not consider lack of knowledge of the policies to be a defense for violating them. Full information is found in the course syllabus– please review it at this time.