Advancement Programming Exam
February 10, 2012

No texts, notes, or Internet access are allowed on this exam. Please read these instructions **CAREFULLY** and **COMPLETELY** before starting.

Your desktop contains a folder "W12APE". That directory contains the following items:

- 4 sub-folders titled: **DataAbstraction, LinkedList, Recursion**, and **General**.
- Each of these folders contains a programming problem that you must address. The LinkedList folder and the Recursion folder contain subfolders.
    - The subfolders in LinkedList are **Part1 – clear_toString_addAll**, **Part2 – addOrdered**, and **Part3 – removeAll**.
    - The subfolders in Recursion are **Part1 – LinkedList** and **Part2 – Ackermann**.
- You do not need to work in any particular order each of the folders can be completed independently from the others. If you are stuck, please free to try the other parts. NOTE: There are some .class files in the folders. **ABSOLUTELY DO NOT** delete these files or you will never get your code to compile. Leave the .class files intact. Also do not modify or attempt to examine any of the .class files.
- Each folder has a file named **output.txt** that illustrates the proper output your program should produce.
- Only modify the source files you are told to for each part. Any other modifications will result in 0 points for that part.
- All work you do on the exam must be done from inside the W12APE folder.

If you are unsure of anything you are being asked to do, seek help from the exam administrator.

DataAbstraction:

- You will need to create 3 classes for this section. The 3 classes are **Undergrad.java**, **Grad.java** and **IDSort.java**.
- Generate two subclasses (**UnderGrad.java** and **Grad.java**) that extend the superclass (**Student.java**). The subclass **UnderGrad** does not contain any unique fields. The subclass **Grad** will be passed an additional parameter, which is the extra per credit fee for a graduate student. You must create a private field in **Grad** to hold this value.
    - These subclasses explicitly invoke the superclass's constructor. See the **Tester** class for what is passed to each constructor.
    - **UnderGrad** and **Grad** also supply the concrete method to override the abstract method that is within **StudentInterface**. The abstract method header is **public abstract double calcFees()**.
        - For **UnderGrad** the fees are calculated by multiplying the number of credits by the cost per credit. The cost per credit can be found as a **public static final double** named **PER_CREDIT** located within the **StudentInterface**.
        - For **Grad** the fees are calculated by multiplying the number of credits by the cost per credit and then adding the additional fees. The additional fees are calculated by multiplying the extra per credit fee by the number of credits.
    - The **Student** class contains a **toString()**that returns a String containing the ID, and the number of credits. You will need to write a **toString()** method for each subclass that calls the **toString()**

from **Student** and then includes the costs/fees <u>on a separate line</u>. The cost should be preceded by the label "`Fees:` ". In addition, "Under Graduate" or "Graduate" is added to the string, on its own line, after the fees. Don't worry about the units, formatting the fees, or minor rounding errors.

- You are provided an abstract **Student** class. A **Student** contains an **int** for the ID, an **int** for the credits. A **compareTo** method is provided that defines the <u>natural order</u> as the number of credits first, and then the ID number.

- You must write the **compare** method for the **IDSort** class. Note: The **IDSort** class implements **Comparator**. It must also ensure that at compile time two **Student** objects are passed to the **compare** method. Students will be compared based on the ID. See below for the API description of the **compare** method in the **Comparator** interface.

  int compare(Student o1, Student o2)

  Compares its two arguments for order.

- Execution of the tester file should produce the output shown in the output file. NOTE: Do ***not*** attempt to modify **Tester.java**.

<u>LinkedList</u>:

There are 3 subfolders within the LinkedList Folder.

- Preliminary notes for each subfolder: There are two fields (head and size) within the **LinkedList** class. When working with the methods you must write, you will utilize these two fields as necessary. In addition, the **LinkedList** class contains a **dummy head node**. Keep this in mind when writing your methods, as well. Also a reminder that you should not modify **Tester.java**. Furthermore, be sure and review the output file (**output.txt**) to see what was produced from running the tester. Do not delete any **.class** files in any of the folders.

- <u>Part 1</u>: Write the **clear()**, **toString()** and **addAll()** methods for **LinkedList.java**. Follow any additional instructions provided in the source file to aid you in building these methods properly. The **addAll** method takes a **Collection** as a parameter. In your case that **Collection** will be an Integer [].

  boolean addAll(Integer [] c)

  Appends all of the elements in the specified Integer array to the end of this list, in the order that the items are in the array.

  The **addAll** method returns true if the list was modified or false otherwise. HINT: You may want to write an add method which is equivalent to an addLast method as a helper.

- <u>Part 2 - addOrdered</u>: Write the **addOrdered** method for **LinkedListA.java**. It assumes the existing list is already in ascending order. The type passed into **addOrdered** must be **Comparable**.

- Part 3 - removeAll: Write the **removeAll** method for **LinkedListB.java**. The method takes a Integer [] as a parameter. It should remove the elements from the linked list that match values found in the array. **NOTE**: You are guaranteed no duplicate elements in the list. It should return **true** if the list is modified, **false** otherwise. If Integer [] is **null** then throw an **IllegalArgumentException**. HINT: You may want to write a remove method as a helper.

Recursion:

There are 2 subfolders within the Recursion Folder. For each problem, see the source code for additional instructions to help you solve the problems specified.

- Part 1 - LinkedList: Write the **subListReverse(**`fromIndex, toIndex`**)** method for **LinkedListC.java**. This method should print the contents of the list fromIndex toIndex (inclusive) in REVERSE order. You may iterate to the fromIndex then you must utilize recursion to find the toIndex. You are expected to write a private helper method to aid your recursion: that method has been stubbed out for you. Follow the comments at the top of the **LinkedListC** file to aid you in accomplishing this task. Do **NOT** attempt to modify **Tester.java**. Do **NOT** delete **any of the .class files** from the folder. Check the output file (**output.txt**) to see what is produced from running the driver.

- Part 2 – Ackermann: Write the recursive method to compute the Ackermann value.

  The Ackermann function is the simplest example of a well-defined total function, which is computable but not primitive recursive, providing a counterexample to the belief in the early 1900s that every computable function was also primitive recursive (Dötzel 1991). It grows faster than an exponential function, or even a multiple exponential function.

  The Ackermann function $A(x, y)$ is defined for integer x and y by

$$A(x,y) \equiv \begin{cases} y+1 & \text{if } x = 0 \\ A(x-1, 1) & \text{if } y = 0 \\ A(x-1, A(x, y-1)) & \text{otherwise.} \end{cases}$$

  Do NOT attempt to modify **Tester.java**. Do **NOT** delete **any of the .class files** from the folder. Check the output file (**output.txt**) to see what is produced from running the driver.

General:

Follow the directions given in the comments in **Tester.java**. Modify the **Tester.java** file to:

- Note you will need to write the full method header and body. Please carefully examine the main method and the comments at the top of where each method should be.

- **openInputFile:** Connect a **Scanner** to the file name specifie. The method accepts a String representing the filename. Handle the **FileNotFoundException** exception with a try/catch. If a **FileNotFoundException** is caught, then as long as the file is not opened, prompt the user for a new filename and attempt to open the file.

- **openOutputFile:** Connect a **PrintStream (or PrintWriter if you prefer)** to the filename specified: print an error message and exit the program on failure. You must use a try/catch to handle the failure.

- **closeInputFile:** Disconnect from the input source

- **closeOutputFile:** Disconnect from the output file

- Write a **read()** method to return the ***entire*** contents of the file as a String. If the file is empty return the string "File is Empty". The **read()** method will read to end-of-file and then return the string. Append a carriage return to each line read from the file.

- Write a **write()** method to print a character string to the **PrintStream** object passed in.

Closing Notes:

- Please logoff your machine when done. Do not turn your machine off.
- Please return these instructions on your way out
- For scratch paper use the back pages of this document. If you need more, ask the exam administrator.
- You can log into the ape website to see your scores once they are posted. It typically takes 2 weeks for this to happen.