COP2800 Homework #2 Assignment Spring 2013

YOUR NAME:_____ **DATE:** _____

LAST FOUR DIGITS OF YOUR UF-ID: ___ ___ ___ ___ **Please Print Clearly (Block Letters)**

Date Assigned: 31 January 2013 IN CLASS

Date Due: 15 February 2013 E-SUBMISSION of Parts II and III

This homework assignment must be completed by you alone. You may not copy from others, and you may not copy code from the Internet, textbook, or other sources.
However, you may study with others or read your textbook to determine general solutions. Then you must complete the problems as your own work, not copying others' work.

Questions about this homework should be addressed to your TA first. You can find your TA's email, office hours, etc. at the class website: http://www.cise.ufl.edu/~wchapman/COP2800/officehours.html

---

This homework has three parts: (I) Vocabulary Questions, (II) Regular Program, (III) Advanced Program. There is no penalty for guessing.

Part I. Vocabulary Questions **[10 points total]**

Vocabulary: (terms you need to know to discuss the subject intelligently) – Define the following terms using 1-3 sentences (and a diagram, if needed): **[2 points each]**

a. *Datatypes* (in Java)
b. *Typecasting* (in Java)
c. *Compilation*
d. *Java Virtual Machine* (JVM)
e. *Subclass / Superclass* (in Java)

Use your text editor (Notepad++) to generate a file called "Part1.txt". Include this file in the ZIP file along with the code for Parts II and III.

You ***must*** have in the upper right-hand corner: (i) "COP2800-S13-HW2-PartI", (ii) your name, and (iii) last four digits of your UFID.

Part II. Regular Program **[20 points total]**

TASK: Create a Java Program that filters numbers to determine what interval they are in, how many numbers per interval; and prints the interval bounds and number counts. This is designed to make you think, so all the code is not provided. You have to do more work here…

PROGRAMMING PROCEDURE:

(1) Use the following pseudocode (this is not Java code) as the basis for making a Java ***Class*** called **NumberFilter**, with method ***FindBin***. Note that comments are in green typeface, and reserved keywords are in bold blue typeface, and bold red typeface shows where you should insert your name and code. The text that you will output to the screen is shown in brown typeface.

(2) Enter the Java code that you make from the example below.

(3) Save your code in file "NumberFilter.java" (save frequently to avoid work loss), then compile using the Java tools that you downloaded to your laptop computer. Make sure it runs.

```
// Pseudocode for COP2800 Assignment #2:  Number Filtering Program
// Objective:  Filter a number into one of N_BINS bins(number line intervals)
// Designed and Coded by:  <your name>      Date:   15 Feb 2013
```

➔ Declare a public class called **`NumberFilter`**

Ex:
```
public class NumberFilter {
public NumberFilter ()
{
}
```

➔ Make an **integer array** of bins structured as `Bins[1..N_BINS, 3]`

Ex:
```
int Nbins = 3;
float[][] Bins = new float[Nbins][3];
```

- This array should be available to all subclasses and their methods in the class *NumberFilter* .
- The first dimension of the array (from 1 to N_{BINS}) indexes the bin, e.g., with variable *i*
- The next dimension of the array references the lower bound (LB), upper bound (UB) and count of numbers in the i^{th} bin (count).
- For Part II of this assignment, the array values will be as follows:

Lower Bound	Upper Bound	Initial Value of "count"
`Bins[0,0] = -10.0f;`	`Bins[0,1] =  0.0f;`	`Bins [0,2] = 0.0f;`
`Bins[1,0] =  1.0f;`	`Bins[1,1] = 10.0f;`	`Bins [1,2] = 0.0f;`
`Bins[2,0] =  11.0f;`	`Bins[2,1] = 25.0f;`	`Bins [2,2] = 0.0f;`

➔ Declare a public **`int`** method for the *NumberFilter* class called *FindBin*

- This method will accept a number (say, *x*) as an argument

Ex:
```
public static int FindBin(float x) { …code here… }
```

- This method will return the integer bin index `bin_number` that the number `x` fits into
- The method will be structured and its code will work as shown in the follow p-code:

Ex:
```
// Step 0: Local variable declarations
      boolean     found = false;
      int         i, bin_number;

// Step 1: Copy the argument of the method
      x =         <argument_of_the_method_FindBin> ;

// Step 2: Loop through all the bins
      for i = 0 to (N_BINS - 1) do:
      {     if (x >= Bins[i, 1]) && (x <= Bins[i, 2])
                  {     Bins[i, 3]++;
                        found = true;
                        bin_number = i;  }
      }
```

```
// Step 3: If the number is found, then output the bin number
//          and output the bin count, then return
      if (found)
            {System.out.println("Bin number=", bin_number);
             System.out.println("Bin count=",
                                        Bins[bin_number, 3]); }
         else
            {System.out.println("Number ", x," not in any bin");}
      return bin_number;
```

(In this case, the bin count will be one (1), because we have tested only one number.)

➔ Make a test routine (Test.java) that will test the correct operation of the *NumberFilter* class with method *FindBin* (this is up to you to design and code).

After you have designed, entered, compiled, and tested your code, then do the following:

(4) Make sure your code runs correctly by using multiple test cases.

(5) Submit your code electronically with your solution to Part III, as described below.

Part III. Advanced Program [40 points total]

Programming with Arrays and Typecasting. Perform the following steps:

(1) Create the following structure with *BetterNumberFilter*, and a *Test2* class, as described below.

Subclass: *BetterNumberFilter* is a subclass of the class *NumberFilter* that extends *NumberFilter* and uses the array Bins from Part II.

Method 1: *FindBin* You do not need to redefine this method, as it is inherited from *NumberFilter* that you developed in Part II.

Method 2: *ScanArray* with *method* ***Scan*** *that loops through a float input array* ***a,*** and applies *FindBin* to each value in the input array. Pseudocode for this method follows:

```
public static void ScanArray(float [] a) {
    int j,result
    for j = 0 to a.length do:
    {    result = FindBin(a[j]);}
}
```

Method 3: *PrintOutput* loops through the number of bins and prints the count of numbers in each bin (i.e., `Bins[i,3]`).

Class: *Test2* (i) inputs an array to *ScanArray*, (ii) calls *ScanArray* to scan through the values in the array, where *ScanArray* applies *FindBin* to each value, accumulating the number of counts per

interval. (iii) After the loop exits, then *Test2* calls *PrintOut* to print the bin index (e.g., 1 through N_{BINS}) and the count of numbers that were found to be in that bin (i.e., `Bins[i,3]`).

(2) Be careful about the way your methods are invoked… [*Hint*: Remember how we invoked `class.method` in Assignment #1.]

(3) Document your code fully (as in the example in Part II). Compile your code and get it running.

(4) Document your code fully (as in the example in Part II). Compile your code and get it running, then test it thoroughly to be sure it works before you submit your code.

Part IV. Extra Credit (write on paper, submit in class) **[10 points each]**

EC-1. What function does the code described in Part III perform? Describe exactly how it works (full credit only for detailed description).

EC-2. What limitations does the *Bins* array in Part III impose on detecting numbers? (*Hint:* If we were sorting integers into bins instead of using floats as input, how would this question be answered?) Under what **three conditions** would the program not sort all floating point numbers (i.e., real numbers) with the existing *Bins* array?

Electronic Submission of Parts II and III. Put all files you created in Parts I through III in a single ZIP file. Your ZIP file should contain all the files specified in Parts I through III. Submit this ZIP file electronically per the instructions at:

http://www.cise.ufl.edu/~wchapman/COP2800/submit

Grading:

Code does not compile or run	= 0 points.
Code compiles but does not run	= < 20 percent of points.
Code runs but wrong results	= 21 to 50 percent of points.
Code runs with correct results but no documentation (e.g., green comments in Part II)	= 51 to 70 percent of points.
Code compiles and runs, correct results, documentation present	= 71 to 100 percent of points.