**CS 116 Spring 2023 Lab #04**

Due: **SUNDAY**, **February 26, 2023 11:59 PM CST**

Points: **20**

**Instructions:**

1. Use this document template to report your answers. Enter all lab partner names at the top of first page.
2. You don’t need to finish your lab work during the corresponding lab session.
3. Name the complete document as follows:

LastName\_FirstName\_CS116\_Lab04\_Report.doc

1. Submit the final document to Blackboard Assignments section before the due date. No late submissions will be accepted.
2. **ALL lab partners need to submit the same report. Only one submission will be graded.**
3. **ALL lab partners need to submit a report, even it is the same document.**

**Objectives:**

1. (5 points) Design, implement, and test a program that require arrays.
2. (6 points) Design, code and test a user-defined class using inheritance.
3. (2 points) Design and implement a new class.
4. (7 points) Demonstrate the ability to use loops, arrays and multiple classes to solve a problem.

**Problem 1 [5 points]:**

Write a program that:

* asks the user how many exam scores there are (**and verifies that the user entered a positive integer**), **[0.5 out of 5 points]**
* prompt the user for each **real-valued** score, one by one (as shown below in **Sample program behavior / output**: box), **[2 out of 5 points]**
* Calculate and output the average of the scores and count and output how many scores are greater than the average (as shown below in **Sample program behavior / output**: box). **[1.5 out of 5 points]**

|  |
| --- |
| **Sample program behavior / output**: |
| How many exams scores do you have to enter? 5  Enter score #1: 95.0  Enter score #2: 92.0  Enter score #3: 68.0  Enter score #4: 72.0  Enter score #5: 70.0  The average score is: 79.4  There are 2 scores larger than the average. |

Provide basic design of your program (inputs/outputs, their format, valid ranges, etc. along with all necessary computation steps) in the box below **[0.5 out of 5 points]**:

|  |
| --- |
| **Program design**: |
| * Prompt the user to enter the number of exam scores. * Validate that the input is a positive integer. * Create an empty list to hold the exam scores. * Use a loop to prompt the user for each exam score and append it to the list. * Use another loop to count the number of scores greater than the average. * Display the average score and the number of scores greater than the average. * Calculate the average of the exam scores.   Inputs:   1. Number of exam scores (positive integer) 2. Exam scores ( array ) (real numbers)   Outputs:   1. Average score (real number) 2. Number of scores greater than the average (integer)   Valid ranges:   1. Number of exam scores: 1 to infinity 2. Exam scores: 0 to 100 |

Complete the Test Plan table below (the number of test cases is up to you, but should be fairly exhaustive) **[0.5 out of 5 points]**:

|  |  |  |  |
| --- | --- | --- | --- |
| **Test plan** | | | |
| **Test case** | **Sample data** | **Expected result** | **Verified?** |
| No exam scores | totalExamScores = 0 | The value given is incorrect! Try again. | Yes |
| Negative Number of Exam Scores | totalExamScores = -4 | The value given is incorrect! Try again. | Yes |
| 3 Exams | totalExamScores =3  examScores = {75,80,85.} | The average score is: 80.3  There are 2 scores larger than the average | Yes |
| Negative Exam Score | totalExamScores = 1  examScores = { -60} | The average score is: 0.0  There are 0 scores larger than the average | Yes |
| No Greater than the average scores | totalExamScores = 2  examScores = { 60 , 65.5} | The average score is: 63.0  There are 0 scores larger than the average | Yes |

**Problem 2 [6 points]:**

A mathematical vector possesses a definite initial and terminal point. Write a class called Vector that will represent a 2D vector **with the initial point at the origin of the coordinate system**. Your class should follow this specification

* It has two fields: terminal point coordinates **[0.25 pt]**
* Terminal point coordinates **can only be set once via class constructor(s)**. You can assume terminal point coordinates are of type int, **[0.25 pt]**
* If terminal point coordinates are not specified, set both to zero, **[0.25 pt]**
* There are five accessor methods: **[1.25 pt, 0.2 pt for each method]**
* getStartPointX(),
* getStartPointY(),
* getEndPointx(),
* getEndPointY(),
* getLength() which calculates and returns vector length,
* There are no mutator functions.

Now, **using inheritance**, create another class called BoundVector, that will also allow to specify vector initial point coordinates (for Vector class it was (0, 0)). Your second class should follow the following specification:

* It has four fields: initial and terminal point coordinates (make sure inheritance is used properly and there is code reuse involved), **[0.5 pts]**
* Create constructors:
* No-argument constructor, **[0.5 pts]**
* “Only terminal point” constructor, **[1 pt]**
* “Initial and terminal point” constructor, **[1 pt]**
* Modify/override superclass methods that need to change, if any **[1 pt]**.

**Problem 3 [2 points]:**

Write a class called StudentID that will model your Illinois Tech HawkCard:



Your StudentID class should:

* store the following information (**choose appropriate data type for each**) **[0.5 pts]**:
  + institution name (which will always be “Illinois Tech”),
  + student’s A number (including “A”),
  + student’s first name,
  + student’s last name,
  + NOTE 1: use Java mechanisms to **ensure that data listed above will be IMPOSSIBLE to change once set**,
  + NOTE 2: make sure that (some) **data is not unnecessarily duplicated** (static!),
* provide **information hiding** **[0.25 pts]**,
* have a non-default parameterized constructor that initializes all student data (three values) at once **[0.5 pts]**:
  + if provided data is **invalid** (either null or empty string [tip: use String class’s equals() method]), set ALL student information to “INVALID”,
* have a non-default non-parameterized constructor that will set ALL student information to “INVALID” **[0.25 pts]**,
* NOT have any setters / mutators,
* have getters / accessors for ALL attributes **[0.25 pts]**.
* have a toString() method that will display ID information as shown on the example below **[0.25 pts]**.

Here’s a test driver / app class (**you are not allowed to modify it**) for your StudentID class / objects:

|  |
| --- |
| IDApp class: |
| public class IDApp {  public static void main (String [] arguments){  StudentID studentA = new StudentID("A00000001", "Jane", "Doe");  StudentID studentB = new StudentID("A00000002", "John", "Doe");  StudentID studentC = new StudentID("", "Janet", "Smith");  StudentID studentD = new StudentID("", "", "Smith");  StudentID studentE = new StudentID("", "", "");  StudentID studentF = new StudentID("A00111112", "Cate", "Blanchett");  StudentID studentG = new StudentID("A10111115", "Keanu", "Reaves");  StudentID studentH = new StudentID();    System.out.println("A:");  System.out.println(studentA);  System.out.println("B:");  System.out.println(studentB);  System.out.println("C:");  System.out.println(studentC);  System.out.println("D:");  System.out.println(studentD);  System.out.println("E:");  System.out.println(studentE);  System.out.println("F:");  System.out.println(studentF);  System.out.println("G:");  System.out.println(studentG);  System.out.println("H:");  System.out.println(studentH);  }  } |

If your StudentID class is complete and you compile and run the driver class, you should see the following output:

|  |
| --- |
| IDApp class output: |
| A:  Illinois Tech  Student ID  Jane  Doe  A00000001  B:  Illinois Tech  Student ID  John  Doe  A00000002  C:  Illinois Tech  Student ID  INVALID  INVALID  INVALID  D:  Illinois Tech  Student ID  INVALID  INVALID  INVALID  E:  Illinois Tech  Student ID  INVALID  INVALID  INVALID  F:  Illinois Tech  Student ID  Cate  Blanchett  A00111112  G:  Illinois Tech  Student ID  Keanu  Reaves  A10111115  H:  Illinois Tech  Student ID  INVALID  INVALID  INVALID |

**Problem 4 [7 points]:**

Your task is to write a class that will imitate a simple ID database (a database for StudentID class [developed in Problem 3] objects) that:

* 1. has the following attributes / fields (select appropriate data types, access modifiers, and static/non-static yourself): **[1 out of 7 points]**
     1. numberOfIDs, that will hold the CURRENT number of IDs in the database (initialize to zero),
     2. MAX\_DATABASE\_SIZE, that sets the limit of IDs that this database can hold (**cannot be changed once set**),
     3. an array of StudentID object references called IDList,
  2. has two constructors: **[1 out of 7 points, 0.5 points for each constructor]**
     1. non-parameterized constructor that will:
        1. set the MAX\_DATABASE\_SIZE attribute / field to 5,
        2. initialize the array,
     2. parameterized constructor that will set the value of MAX\_DATABASE\_SIZE to provided value/parameter/argument **IF provided value is valid/legal**. Otherwise set it the same way as in the non-parameterized constructor,
  3. has accessor / getter method for the numberOfIDs attribute **[1 out of 7 points]**
  4. has an add method that allows you to add a new student ID to your database that: **[2 out of 7 points]**
     1. prevents from adding “null IDs”,
     2. prevents from exceeding database maximum size,
     3. prevents from adding IDs with “INVALID” values for ANumber,
     4. updates all corresponding attributes / fields accordingly,
     5. returns true if adding was successful (ID added), and false otherwise (ID could not be added),
  5. has a method listNames that will use StudentID object methods to display selected ID information in sequence (see example below). If the database is empty, it should display “ERROR: empty database.” message, **[1 out of 7 points]**
  6. has a method toString that will display database summary as shown in example below. **[1 out of 7 points]**

If your StudentID and IDDatabase classes are implemented correctly, the following application class:

|  |
| --- |
| IDDatabaseApp class: |
| public class IDDatabaseApp {  public static void main(String [] args){  // Instantiate new database  IDDatabase myDBase = new IDDatabase(6);    // create and add some IDs to it  StudentID ID1 = new StudentID("A00000001", "Jane", "Doe");  myDBase.add(ID1);  StudentID ID2 = new StudentID("A00000002", "John", "Doe");  myDBase.add(ID2);  ID1 = new StudentID("", "Janet", "Smith");  myDBase.add(ID1);  ID2 = new StudentID("", "", "Smith");  myDBase.add(ID2);  ID1 = new StudentID("A00111112", "Cate", "Blanchett");  myDBase.add(ID1);  StudentID ID3 = new StudentID("A10111115", "Keanu", "Reaves");  myDBase.add(ID3);  StudentID ID4 = new StudentID("A99999999", "Kanye", "West");  myDBase.add(ID4);  StudentID ID5 = new StudentID("A99456678", "Michael", "Jordan");  myDBase.add(ID5);  StudentID ID6 = new StudentID();  myDBase.add(ID6);  StudentID ID7 = new StudentID("A19436678", "Cristiano", "Ronaldo");  myDBase.add(ID7);    // list all names  myDBase.listNames();    // Display database "summary"  System.out.println(myDBase.toString());  // NOTE: it's the same as System.out.println(myDBase);  }  } |

Should generate this output (note how listNames and toString methods work):

|  |
| --- |
| IDDatabaseApp class output: |
| ID 0: Jane Doe  ID 1: John Doe  ID 2: Cate Blanchett  ID 3: Keanu Reaves  ID 4: Kanye West  ID 5: Michael Jordan  Database:  Illinois Tech  Student ID  Jane  Doe  A00000001  Illinois Tech  Student ID  John  Doe  A00000002  Illinois Tech  Student ID  Cate  Blanchett  A00111112  Illinois Tech  Student ID  Keanu  Reaves  A10111115  Illinois Tech  Student ID  Kanye  West  A99999999  Illinois Tech  Student ID  Michael  Jordan  A99456678 |