[redacted]

SBCS
GLOBAL LEARNING INSTITUTE

[redacted]	[redacted]
[redacted]	[redacted]
[redacted]	[redacted]
[redacted]	[redacted]
[redacted]	[redacted]
[redacted]	[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

START OF SECTION A - TOTAL 100 MARKS

Please answer all three (3) questions in this section

Question 1 - Sum of Integers within a Range (30 Marks)

You must design, develop, test and implement a *command line* Java program with the following specifications:

S1) The program must be named **SumIntRange** and will always take only 3 *command line* arguments.

S2) The 1^{st} *command line* argument will always be 1 of the 3 lower case strings: **-all**, **-even** or **-odd**.

S3) The 2^{nd} and 3^{rd} *command line* arguments will always be positive integers (i.e greater than zero) with the 2^{nd} *command line* argument being strictly less than the 3^{rd} *command line* argument.

```
> java SumIntRange -all 1 6
21
> java SumIntRange -even 1 6
12
> java SumIntRange -odd 1 6
9
```

Explanation of Execution Results:

E1) The 1^{st} invocation of SumIntRange specifies a calculation of the sum of **all** the integers between **1** and **6** *inclusive*. Thus, the output is **21**.

E2) The 2^{nd} invocation of SumIntRange specifies a calculation of the sum of all the **even** integers between **1** and **6** *inclusive*. Thus, the output is **12**.

E3) The 3^{rd} invocation of SumIntRange specifies a calculation of the sum of all the **odd** integers between **1** and **6** *inclusive*. Thus, the output is **9**.

Figure 1 - Three (3) invocations of the **SumIntRange** program from the *command line*

You must create the files **SumIntRange.java** and **SumIntRangeTester.java** as part of your solution to this question which will consist of the deliverables specified below in the sections **A** to **D**.

A - Design (5 Marks)

Deliverable **A1** - A detailed IPO chart for the **sumIntegersInRange** function, which implements the functionality shown in **Figure 1** above. The **sumIntegersInRange** function must return a single *integer* and must take 3 parameters: a *string*, an *integer* and an *integer* in the same order as the arguments specified on the command line. You must clearly show all steps in the algorithm of the **sumIntegersInRange** function especially how the 3 parameters are used to calculate the final integer return value. See specifications **S2** and **S3** above for further details.

B - Unit Testing (9 Marks)

Deliverable **B1** - The file **SumIntRangeTester.java** which must contain at least twelve (12) unit tests for the **sumIntegersInRange** function. The only other file you are allowed to use is **TestUtils.java** that was given in class.

C - Development (12 Marks)

Deliverable **C1** - The file **SumIntRange.java** which contains your solutions to the **sumIntegersInRange** function and the **main** procedure. The development of the **main** procedure must ensure that the **SumIntRange** program works as specified in **Figure 1** above. You are not allowed to use any type of array or list variable *besides* **String [] args**. You are not allowed to use any standard or third party Java utilities. All code must be your own. No other Java files must be used.

D - Implementation (4 Marks)

Deliverable **D1** - A screen shot which shows the output of running the **SumIntRangeTester** program from the *command line without any command line arguments*. This screen shot must show the final results of running all tests.

Deliverable **D2** - A screen shot which shows multiple invocations of the **SumIntRange** program on the command line. You must have one invocation per unit test and each invocation of the **SumIntRange** program on the command line must use command line arguments that exactly match the arguments of the corresponding unit test in **SumIntRangeTester.java**.

Question 2 - Stacking Bricks (30 Marks)

You must design, develop, test and implement a *command line* Java program with the following specifications:

S1) The program must be named **StackBricks** and will always take only 4 positive integer *command line* arguments.

S2) The 1^{st} and 2^{nd} and 3^{rd} *command line* arguments represent the dimensions of a brick which is in the shape of cuboid. The 4^{th} *command line* argument represents the number of bricks.

S3) The **StackBricks** program must display the minimum and maximum possible height of the stack of bricks.

```
> java StackBricks 1 2 3 3
Min Height:  3, Max Height:  9
> java StackBricks 4 6 2 5
Min Height:  10, Max Height:  30
> java StackBricks 5 4 3 7
Min Height:  21, Max Height:  35
```

Explanation of Execution Results:

E1) The 1^{st} invocation of StackBricks shows that the minimum and maximum heights of **3** bricks, each of dimensions **1x2x3**, are **3** and **9** respectively.

E2) The 2^{nd} invocation of StackBricks shows that the minimum and maximum heights of **5** bricks, each of dimensions **4x6x2**, are **10** and **30** respectively.

E3) The 3^{rd} invocation of StackBricks shows that the minimum and maximum heights of **7** bricks, each of dimensions **5x4x3**, are **21** and **35** respectively.

Figure 2 - Three (3) invocations of the **StackBricks** program from the *command line*

You must create the files **StackBricks.java** and **StackBricksTester.java** as part of your solution to this question which will consist of the deliverables specified below in the sections **A** to **D**.

A - Design (5 Marks)

Deliverable **A1** - A detailed IPO chart for the **minimumStackHeight** and **maximumStackHeight** functions, which implements the functionality shown in **Figure 2** above. The **minimumStackHeight** and **maximumStackHeight** functions each take 4 positive integer parameters (the first 3 integer parameters represent the dimensions of each brick and the last integer parameter represents the number of bricks in the stack). The **minimumStackHeight** function returns an integer representing the minimum possible height of the stack of all bricks. The **maximumStackHeight** function returns an integer representing the maximum possible height of the stack of all bricks. You must clearly show all steps in the algorithms of both functions especially how all the 4 integer parameters are used to determine both the minimum and maximum height of the stack of bricks. See specifications **S2** and **S3** above for further details.

B - Unit Testing (9 Marks)

Deliverable **B1** - The file **StackBricksTester.java** which must contain at least twelve (12) unit tests for the **minimumStackHeight** function, at least twelve (12) unit tests for the **maximumStackHeight** function and at least twelve (12) unit tests for the **getOutput** function (see part C below for the specification of the **getOutput** function). The only other file you are allowed to use is **TestUtils.java** that was given in class.

C - Development (12 Marks)

Deliverable **C1** - The file **StackBricks.java** which contains your solutions to the **minimumStackHeight**, **maximumStackHeight** and **getOutput** functions as well as the **main** procedure. The **getOutput** function will take the same parameters as the **minimumStackHeight** and **maximumStackHeight** functions (as described above) and will return a String representing the message that is actually displayed on the screen. The development of the **main** procedure must ensure that the **StackBricks** program works as specified in **Figure 2** above. You are not allowed to use any type of array or list variable *besides* **String [] args**. You are not allowed to use any standard or third party Java utilities *besides* String.format(). All code must be your own. No other Java files must be used.

D - Implementation (4 Marks)

Deliverable **D1** - A screen shot which shows the output of running the **StackBricksTester** program from the *command line without any command line arguments*. This screen shot must show the final results of running all tests.

Deliverable **D2** - A screen shot which shows multiple invocations of the **StackBricks** program on the command line. You must have one invocation per unit test and each invocation of the **StackBricks** program on the command line must use command line arguments that exactly match the arguments of the corresponding unit test in **StackBricksTester.java**.

Question 3 - Filling Tanks (40 Marks)

You must design, develop, test and implement a *command line* Java program with the following specifications:

S1) The program must be named **FillTank** and will always take only 4 positive integer *command line* arguments.

S2) The 1^{st} and 2^{nd} and 3^{rd} *command line* arguments are the *length*, *width* and *height* respectively of the tank in metres (m).

S3) The tank is in the shape of a cuboid where the *length* and *width* dimensions represent the base area of the tank that rests flat on the ground. The *height* dimension represents the vertical height of the tank. The 4^{th} *command line* argument represents the volume of water to be poured into the tank in metres cubed (m^3).

```
> java FillTank 3 4 5 54
Vacant Height of 0.50
> java FillTank 3 4 5 60
Tank Filled
> java FillTank 3 4 5 70
Overflow of 10
```

Explanation of Execution Results:

E1) The 1^{st} invocation of FillTank shows that pouring $54m^3$ of water into a $3m \times 4m \times 5m$ tank (of volume $60m^3$) leaves a vacant height of $0.50m$ that is *untouched by water*.

E2) The 2^{nd} invocation of FillTank shows that pouring $60m^3$ of water exactly fills a $3m \times 4m \times 5m$ tank (since the tank has a volume of $60m^3$).

E3) The 3^{rd} invocation of FillTank shows that when $70m^3$ of water is poured into a $3m \times 4m \times 5m$ tank (of volume $60m^3$), $10m^3$ of excess water will overflow.

Figure 3 - Three (3) invocations of the **FillTank** program from the *command line*

You must create the files **FillTank.java** and **FillTankTester.java** as part of your solution to this question which will consist of the deliverables specified below in the sections **A** to **D**.

Please note that in **Figure 3** above, when there is a vacant height, a floating-point number must be displayed rounded to two (2) decimal places (the first example) but when there is an overflow, an integer must be displayed (the third example).

A - Design (7 Marks)

Deliverable **A1** - A detailed IPO chart for the **pourWaterIntoTank** function, which implements the functionality shown in **Figure 3** above. The **pourWaterIntoTank** function must return a single *String* and must take 4 positive integer parameters in the same order as those 4 positive integer arguments specified on the command line. You must clearly show all steps in the algorithm of the **pourWaterIntoTank** function especially how the 4 parameters are used to calculate the final String return value. See specifications **S2** and **S3** above for further details.

B - Unit Testing (11 Marks)

Deliverable **B1** - The file **FillTankTester.java** which must contain at least twelve (12) unit tests of the **pourWaterIntoTank** function. The only other file you are allowed to use is **TestUtils.java** that was given in class.

C - Development (16 Marks)

Deliverable **C1** - The file **FillTank.java** which contains your solutions to the **pourWaterIntoTank** function and the **main** procedure. The development of the **main** procedure must ensure that the **FillTank** program works as specified in **Figure 3** above. You are not allowed to use any type of array or list variable *besides* **String [] args**. You are not allowed to use any standard or third party Java utilities *besides* String.format(). All code must be your own. No other Java files must be used.

D - Implementation (6 Marks)

Deliverable **D1** - A screen shot which shows the output of running the **FillTankTester** program from the *command line without any command line arguments*. This screen shot must show the final results of running all tests.

Deliverable **D2** - A screen shot which shows multiple invocations of the **FillTank** program on the command line. You must have one invocation per unit test and each invocation of the **FillTank** program on the command line must use command line arguments that exactly match the arguments of the corresponding unit test in **FillTankTester.java**.

END OF SECTION A

START OF BONUS SECTION - TOTAL 100 MARKS

You may attempt none, one or all questions in this section

Bonus Question 1 - Median Integer (50 Marks)

You must design, develop, test and implement a *command line* Java program with the following specifications:

S1) The Java program must be named **MedianInteger** and will always take only 3 positive integer *command line* arguments.

S2) The **MedianInteger** program must display the *median* of the 3 integers which is the middle integer after rearranging the 3 integers from left to right in ascending order.

```
> java MedianInteger 3 7 5
5
> java MedianInteger 4 2 6
4
> java MedianInteger 9 3 3
3
> java MedianInteger 5 2 5
5
```

Explanation of Execution Results:

E1) The median of the 3 integers **3**, **7** and **5** is **5**. This is because after rearranging the the 3 integers from left to right in ascending order, we get **3**, **5**, **7** and the middle integer is **5**.

E2) The median of the 3 integers **4**, **2** and **6** is **4**. This is because after rearranging the the 3 integers from left to right in ascending order, we get **2**, **4**, **6** and the middle integer is **4**.

E3) The median of the 3 integers **9**, **3** and **3** is **3**. This is because after rearranging the the 3 integers from left to right in ascending order, we get **3**, **3**, **9** and the middle integer is **3**.

E4) The median of the 3 integers **5**, **2** and **5** is **5**. This is because after rearranging the the 3 integers from left to right in ascending order, we get **2**, **5**, **5** and the middle integer is **5**.

Figure 4 - Four (4) invocations of the **MedianInteger** program from the *command line*

You must create the files **MedianInteger.java** and **MedianIntegerTester.java** as part of your solution to this question which will consist of the deliverables specified below in the sections **A** to **D**.

A - Design (10 Marks)

Deliverable **A1** - A detailed IPO chart for the **median** function, which implements the functionality shown in **Figure 4** above. The **median** function must return a single *integer* and must take 3 positive integer parameters in the same order as the 3 positive integer arguments specified on the command line. You must clearly show all steps in the algorithm of the **median** function especially how the 3 parameters are used to calculate the final integer return value. See specification **S2** above for further details.

B - Unit Testing (12 Marks)

Deliverable **B1** - The file **MedianIntegerTester.java** which must contain at least sixteen (16) unit tests within the **main** procedure. The only other file you are allowed to use is **TestUtils.java** that was given in class.

C - Development (20 Marks)

Deliverable **C1** - The file **MedianInteger.java** which contains your solutions to the **median** function and the **main** procedure. The development of the **main** procedure must ensure that the **MedianInteger** program works as specified in **Figure 4** above. You are not allowed to use any type of array or list variable *besides* **String [] args**. You are not allowed to use any standard or third party Java utilities. All code must be your own.

D - Implementation (8 Marks)

Deliverable **D1** - A screen shot which shows the output of running the **MedianIntegerTester** program from the *command line without any command line arguments*. This screen shot must show the final results of running all tests.

Deliverable **D2** - A screen shot which shows multiple invocations of the **MedianInteger** program on the command line. You must have one invocation per unit test and each invocation of the **MedianInteger** program on the command line must use command line arguments that exactly match the arguments of the corresponding unit test in **MedianIntegerTester.java**.

Bonus Question 2 - Hide the Rectangle (50 Marks)

You must design, develop, test and implement a *command line* Java program with the following specifications:

S1) The Java program must be named **HideRec** and will always take only 4 positive integer *command line* arguments.

S2) The 1^{st} and 2^{nd} *command line* arguments represent the dimensions of the first rectangle. The 3^{rd} and 4^{th} *command line* arguments represent the dimensions of the second rectangle.

S3) The **HideRec** program must indicate if it is possible for one rectangle to completely hide the other rectangle when one edge in the first rectangle is parallel to another edge in the second rectangle.

```
> java HideRec 4 7 2 3
4x7 completely hides 2x3
> java HideRec 2 3 7 4
7x4 completely hides 2x3
> java HideRec 5 2 4 1
5x2 completely hides 4x1
> java HideRec 1 4 5 2
5x2 completely hides 1x4
> java HideRec 3 3 5 5
5x5 completely hides 3x3
> java HideRec 5 5 3 3
5x5 completely hides 3x3
> java HideRec 4 7 8 3
Both Rectangles Visible
```

Explanation of Execution Results:

E1) The 1^{st} invocation of HideRec shows that it is possible for a 4x7 rectangle to completely hide a 2x3 rectangle when the length of the first rectangle (4) is parallel to the length of the second rectangle (2).

E2) The 2^{nd} invocation of HideRec shows that it is possible for a 7x4 rectangle to completely hide a 2x3 rectangle when the length of the first rectangle (2) is parallel to the width of the second rectangle (4).

E3) The 3^{rd} invocation of HideRec shows that it is possible for a 5x2 rectangle to completely hide a 4x1 rectangle when the length of the first rectangle (5) is parallel to the length of the second rectangle (4).

E4) The 4^{th} invocation of HideRec shows that it is possible for a 5x2 rectangle to completely hide a 1x4 rectangle when the length of the first rectangle (1) is parallel to the width of the second rectangle (2).

E5) The 5^{th} invocation of HideRec shows that it is possible for a 5x5 rectangle to completely hide a 3x3 rectangle when the length of the first rectangle (3) is parallel to the length of the second rectangle (5).

E6) The 6^{th} invocation of HideRec shows that it is possible for a 5x5 rectangle to completely hide a 3x3 rectangle when the length of the first rectangle (5) is parallel to the length of the second rectangle (3).

E7) The 7^{th} invocation of HideRec shows that it is not possible for a 4x7 rectangle to completely hide a 8x3 rectangle and it is not possible for a 8x3 rectangle to completely hide a 4x7 rectangle.

Figure 5 - Seven (7) invocations of the **HideRec** program from the *command line*

You must create the files **HideRec.java** and **HideRecTester.java** as part of your solution to this question which will consist of the deliverables specified below in the sections **A** to **D**.

A - Design (10 Marks)

Deliverable **A1** - A detailed IPO chart for the **firstRecHidesSecondRec** function, which will help to implement the functionality shown in **Figure 5** above. The **firstRecHidesSecondRec** function must return a single *boolean* value and must take four (4) positive integer parameters in the same order as those 4 positive integer arguments specified on the command line.

The **firstRecHidesSecondRec** function should return true only if the first rectangle (whose dimensions is given by the first 2 parameters to the function) completely hides the second rectangle (whose dimensions is given by the last 2 parameters to the function) when one side of the first rectangle is parallel to one side of the second rectangle.

You must clearly show all steps in the algorithm of the **firstRecHidesSecondRec** function especially how the 4 parameters are used to calculate the final boolean return value. See specifications **S2** and **S3** above for further details.

B - Unit Testing (12 Marks)

Deliverable **B1** - The file **HideRecTester.java** which must contain within the **main** procedure at least thirty two (32) unit tests for the **firstRecHidesSecondRec** function and at least thirty two (32) unit tests for the **getOutput** function (see part C below for the specification of the **getOutput** function) showing different permutations of the 4 integer arguments with several different sized and equal sized rectangles. The only other file you are allowed to use is **TestUtils.java** that was given in class.

C - Development (20 Marks)

Deliverable **C1** - The file **HideRec.java** which contains your solutions to the functions **firstRecHidesSecondRec** and **getOutput** as well as the **main** procedure. The **getOutput** function will take the same parameters in the same order as the 4 positive integer arguments specified on the command line (as described above) and will return a String representing the message that is actually displayed on the screen. The development of the **main** procedure must ensure that the **HideRec** program works as specified in **Figure 5** above. You are not allowed to use any type of array or list variable *besides* **String [] args**. You are not allowed to use any standard or third party Java utilities *besides* String.format(). All code must be your own. No other Java files must be used.

D - Implementation (8 Marks)

Deliverable **D1** - A screen shot which shows the output of running the **HideRecTester** program from the *command line without any command line arguments*. This screen shot must show the final results of running all tests.

Deliverable **D2** - A screen shot which shows multiple invocations of the **HideRec** program on the command line. You must have one invocation per unit test and each invocation of the **HideRec** program on the command line must use command line arguments that exactly match the arguments of the corresponding unit test within the file **HideRecTester.java**.

END OF BONUS SECTION