CSCI 1101 - 2017
Assignment 2

Complete the assigned tasks and submit a ZIP file containing your finished source code (these are the .JAVA files) on Brightspace.

Your assignment **must be your own work**. You may ask questions of your instructor, course TAs, or Learning Centre TAs. If you discuss the ideas in this assignment with your classmates, **it is your responsibility to make sure you are not plagiarizing** – take no notes during the discussion, wait 30 minutes, and whatever you remember is safe to be used in your own code.

Submission Deadlines (firm):
Due: Saturday July 15, by 11:59pm (midnight)
Late submission (10% penalty): Sunday July 16, by 11:59pm (midnight)

Submissions outside of these given deadlines are **not** accepted.

Before submitting, check that:

1. Your code is properly formatted, includes reasonable comments (including the header comment), and is properly tested for the given problem.
2. Your submission ZIP file contains your **source code** .JAVA files, not your compiled .CLASS files.
3. Your code **compiles** and can be run – even if your program does not do everything perfectly, make sure it compiles before you submit.

Header Comments

Your code should now include header comments for **all** of your class (.java) files. The comment should include the lab/assignment number, the course (CSCI 1101), the name of your program and a short description of the entire class, the date, your name and Banner ID, and a declaration that matches the first page of this document (e.g., whether you received help). See the example below for what a header comment should include:

```
/*Lab1, Question 1 CSCI 1101
   Student.java holds information about a student at Dalhousie in CSCI1101
and
   their grades
   June 29, 2015
   John Smith B00112345
   This is entirely my own work. */

public class Student {
//rest of Code
```

If applicable, your demo class should then also have a similar header:

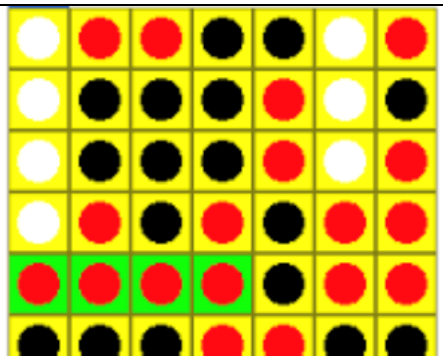
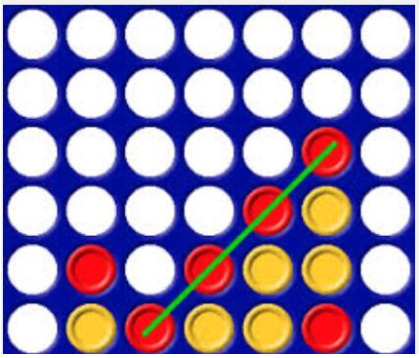
```
/*Lab1, Question 1  - demo class CSCI 1101
   StudentDemo.java is a demo program for the Student class. It creates
student
   objects, and compares different students.
   June 29, 2015
   John Smith B00112345
   I received help with creating Student objects from my TA but the rest is
my
   own work. */

public class StudentDemo {
//rest of Code
```

Question 1
This assignment will simulate a two-player game called Connect 4. In this game, each player takes turns dropping discs into one of 7 columns in a vertical tray. Each column has space for 6 discs to stack vertically. The object of the game is to get 4 of your own discs in a row: vertically, horizontally, or diagonally.

Details of the game are available on Wikipedia:
https://en.wikipedia.org/wiki/Connect_Four

An interactive version of the game can be played here:
http://www.mathplayground.com/connect4.html

The three images below show various stages of the game:

		
The game board partially filled with the discs (or "chips")	A version of the game board, with red winning horizontally.	A version of the game board, with red winning diagonally.

Instead of using a true graphical interface (or GUI), this assignment will simulate the game using only text to represent the game board.

Our Connect 4 board will be 6 rows by 7 colums, and use yellow and red chips (one colour per player). To play, the player will indicate a column, and the game will place their chip at the bottom-most available position in this column. Of course, chips cannot be placed outside the board.

The game should continue until one player wins by connecting four of their chips on the board. If the board becomes full before this happens, the game is a draw. Once each game ends, allow the user to indicate whether they want to play again, or else quit your application. Sample output is given below; your game does not need to match this *exactly*, but it *should* provide the same information in a clear and easily-understood manner.

Along with using concepts of inheritance and aggregation, you may want to use 2d arrays for programming your game. A short tutorial/refresher on this topic is included below.

Your game should use the following UML diagram:

Chip
- colour : String
+ Chip (c : String) : + get/set methods for all the attributes + equals (c : Chip) : boolean + toString ( ) : String

Board
- rows : int - cols : int - board[][] : Chip
+ Board (r : int, c : int) : + get methods for all the attributes + isEmpty (r : int, c : int) : boolean + add (r : int, c : int, chip : Chip) : boolean

Connect4Board
+ Connect4Board ( ) : + add (c : int, colour : String) : boolean + winType ( ) : String + winner ( ) : boolean + toString ( ) : String

A brief description of each class follows:

Chip
Holds information about a Chip object: specifically, its colour (for our Connect 4 game, this should be either red or yellow). The constructor should initialize the colour of the Chip. The equals method should determine whether two Chip objects are equivalent; you should decide what it will mean for two Chip objects to be equivalent. The toString method should return a String representing the colour of the current Chip. Note that as constant values (variables using the final keyword) are not usually specified in UML diagrams, you may add any final variables that would be useful.

Board
This class could be used to implement a variety of generic square-grid game boards. Two private attributes stored the number of rows and columns on the board, and a two-dimensional array stores the actual Chips. The constructor should initialize all the instance variables. Note that arrays of Object references have a value of "null" by default. The isEmpty method returns true if the position given by the row/column parameters does not contain a Chip object, and returns false otherwise. The add method adds the parameter Chip to the given row/column location, so long as this is a valid and empty location; the method returns true if the Chip was successfully added, and false otherwise.

Connect4Board
This class is a subclass of the Board class. The game board for a Connect 4 game should always have 6 rows and 7 columns. The add method should attempt to add a Chip of the given colour in the column indicated by the parameter, returning true if this was successful, and false otherwise. Remember that Chips in Connect 4 are added to a column and should "fall" to the bottom-most available location.

The winType method should return a String that represents whether the current victory was because of a horizontal, vertical, or diagonal match; you should decide on an appropriate return value if there is no current winner. The winner method returns true if either of the players has won the game, either with a vertical, horizontal, or diagonal match. The toString method should return a String that visually represents the current state of the board; see the sample output below for an example of how this might look.

You may add other "helper" methods to this class, at your own discretion.

A third class will be used to actually play the game; previously, this would be called a "demo" class, but here this will be the **Play** class. The Play class should use a Scanner object to collect the names of the two players (assume they are both sitting at and sharing the same keyboard) and collect each of their moves (the column into which they want to add one of their Chips). Create a new Connect4Board to play the game, and print out the full state of the board after each player's move. The game should continue until either one player has won, or until the board is full and no more pieces may be added.

```
Welcome to Connect 4. Please enter your names.
Player 1 name: Bob
Player 2 name: Jane
Bob - you have red chips "R" and you go first.


          1    2    3    4    5    6    7
     1
     2
     3
     4
     5
     6


Please input a column# between 1-7: 1

          1    2    3    4    5    6    7
     1
     2
     3
     4
     5
     6    R


Jane - you have yellow chips "Y" and you go next.

Please input a column# between 1-7: 1

          1    2    3    4    5    6    7
     1
     2
     3
     4
     5    Y
```

```
     6    R

 RED Please input a column# between 1-7: 4

          1    2    3    4    5    6    7
     1
     2
     3
     4
     5    Y
     6    R              R


 YELLOW Please input a column# between 1-7: 3

          1    2    3    4    5    6    7
     1
     2
     3
     4
     5    Y
     6    R         Y    R
RED Please input a column# between 1-7: 4

          1    2    3    4    5    6    7
     1
     2
     3
     4
     5    Y              R
     6    R         Y    R


 YELLOW Please input a column# between 1-7: 2

          1    2    3    4    5    6    7
```

```
    1
    2
    3
    4
    5    Y              R
    6    R    Y    Y    R

RED Please input a column# between 1-7: 4

         1    2    3    4    5    6    7
    1
    2
    3
    4                   R
    5    Y              R
    6    R    Y    Y    R


YELLOW Please input a column# between 1-7: 2

         1    2    3    4    5    6    7
    1
    2
    3
    4                   R
    5    Y    Y         R
    6    R    Y    Y    R

RED Please input a column# between 1-7: 4

         1    2    3    4    5    6    7
    1
```

```
     2
     3                    R
     4                    R
     5    Y    Y          R
     6    R    Y    Y     R
```

RED - Connect 4! Congratulations Bob! You Win in 9 turns.

Play Again? Y/N: Y

Short Tutorial on 2-d arrays

2-d arrays are very similar to 1-d arrays except that they have two subscripts or indices, one representing the row number and the other representing the column number.

For example,

```
int[][] a = new int[5][5];
```

creates a 2-d array with 5 rows and 5 columns.

You can process the 2-d array in a similar manner to that of 1-d arrays. Instead of a single *for* loop, we use a nested *for* loop. For example, the loop

```
for(i=0;i<5;i++)
{
      for(j=0;j<5;j++)
            System.out.print(a[i][j] + "\t");
      System.out.println();
}
```

will print the contents of the array *a* as a 5X5 matrix using tabs for spacing.

The boxes will be numbered as follows:

	Column 0	Column 1	Column 2	Column 3	Column 4
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]
Row 3	a[3][0]	a[3][1]	a[3][2]	a[3][3]	a[3][4]
Row 4	a[4][0]	a[4][1]	a[4][2]	a[4][3]	a[4][4]

If you want to print all the elements in Column 2, for example:

```
for(i=0;i<5;i++)
            System.out.println(a[i][2]);
```

Similarly, the following will print the elements in Row no.3

```
for(j=0;j<5;j++)
            System.out.print(a[3][j] + "\t");
```

The following program creates a 2-d array, reads 25 integers from the keyboard, prints it as a 5X5 matrix and finds the sum of all elements. Note that the 25 integers can be entered all on a single line when they are read from the keyboard. Try it out.

```
import java.util.Scanner;
public class TwoDArray
{
      public static void main(String[] args)
      {
            int[][] a = new int[5][5];
            int i,j, sum=0;
```

```
            Scanner keyboard = new Scanner(System.in);
            for(i=0; i<5;i++)
                  for (j=0;j<5; j++)
                        a[i][j]= keyboard.nextInt();

            for(i=0;i<5;i++)
            {
                  for(j=0;j<5;j++)
                        System.out.print(a[i][j] + "\t");
                  System.out.println();
            }

            for(i=0;i<5;i++)
                  for(j=0;j<5;j++)
                        sum+=a[i][j];
            System.out.println("The sum of all elements is: " + sum);
      }

}
```

Array initializer expression for 2-d arrays

A 2-d array can be created using an array initializer expression. For example:

```
int[][] numbers = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};
```

creates the following array

```
1     2     3     4
5     6     7     8
9     10    11    12
```

Length field in a 2-d array

A 2-d array has a length field that holds the number of rows, and each row has a length field that holds the number of columns.

The following program uses the length fields of a 2d array to display the number of rows, and the number of columns in each row. Try it out.

```
public class Lengths
{
      public static void main(String[] args)
      {
            int[][] numbers = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};
            System.out.println("The number of rows is " + numbers.length);
            for(int index=0; index<numbers.length; index++)
                  System.out.println("The number of columns in row " +
                  index + " is " +      numbers[index].length);
      }
}
```

As with 1-d arrays, the length field only displays the number of components in the array as declared. When using 1-d or 2-d arrays of Objects, the length field will not necessarily tell you the number of instantiated/initialized Objects in that array.