Concordia University
COMP 248 – Fall 2019
Assignment 4 – Edited Nov 20
(Edits in pink on page 6)

Due Date:	By 11:55pm Monday December 2, 2019
(Note: No extensions will be granted as final exam is on Dec 5th)	
Evaluation:	8% of final mark (see marking rubric at the end of handout)
Late Submission:	none accepted
Purpose:	The purpose of this assignment is to practice defining new types/classes and then using them in your program.
CEAB/CIPS Attributes:	Design/Problem analysis/Communication Skills

General Guidelines When Writing Programs:
See previous assignments.

Advice on how to tackle this assignment:

- ***You have 4 weeks to do this assignment. Do not leave it to the last minute.***
- As soon as you have completed the 1st set of slides on classes, write the 1st three classes from part A and test them to make sure they work as you expect them to. You don`t want to be debugging your classes and your driver class all at the same time.
- For part B you will need to have covered classes and arrays of objects. Take the time to plan your solution before starting to program. This means be sure you understand how the game works before your program. Write as detailed an algorithm as you can!!!!!!

- When implementing the driver (`LetsPlay`) do it in chunks. Do not implement the entire game all at once. It will be harder to debug and will be frustrating.
- You will learn a lot from this assignment. Do not waste your time looking for solutions online. It is a made up game. You have all the knowledge from the course content needed to complete this programming assignment.
- Happy programming and of course, Happy "warrioring" (made up word)!

Rules of the game: Details will be supplied in the implementation descriptions.

For this question you will program **Nancy's 3D Warrior Game.** To win this game you need some luck with the dice and a bit of strategy. The first player to land on the top level and last square is the winner.

➔ Refer to the handout ***SampleOutput.pdf*** for a sample run which illustrates the expected behavior of your program while reading the description.

1. **Number of players**: 2
2. **Game Board:** Game board is a 3 dimensional board. Each player starts at level 0 and climbs to the next level, when they have moves across all squares of the current level.

3. **Energy Units**: Each player starts the game with 10 energy units. To move a player must have at least 1 unit, otherwise they are too weak to move. Landing on certain squares will give you extra units, or can have you lose units of energy. Whenever a player rolls a double his/her energy units go up by 2. When a player lands on an already occupied square, they can challenge the player there and possibly gain half of that player's energy units.
4. **The Game Board** is a 3 dimensional board where the 1st dimension represents the number of levels to climb, and the two other values are a specific location in the size x size board at that level, where size is the dimensions of the board on each level.
   For example, if the game board has 3 levels, and each level has a 4x4 board, then the starting position is (0,0) on level 0 and the winning position is (3,3) on level 2 (levels run from 0 to 2). The winner is the one who reaches level 2 location (3,3) first. You move from location (0,0) to (3,3) on any given board before climbing to the next level. The red arrows in Figure 1, show the way a player moves through the squares and levels.

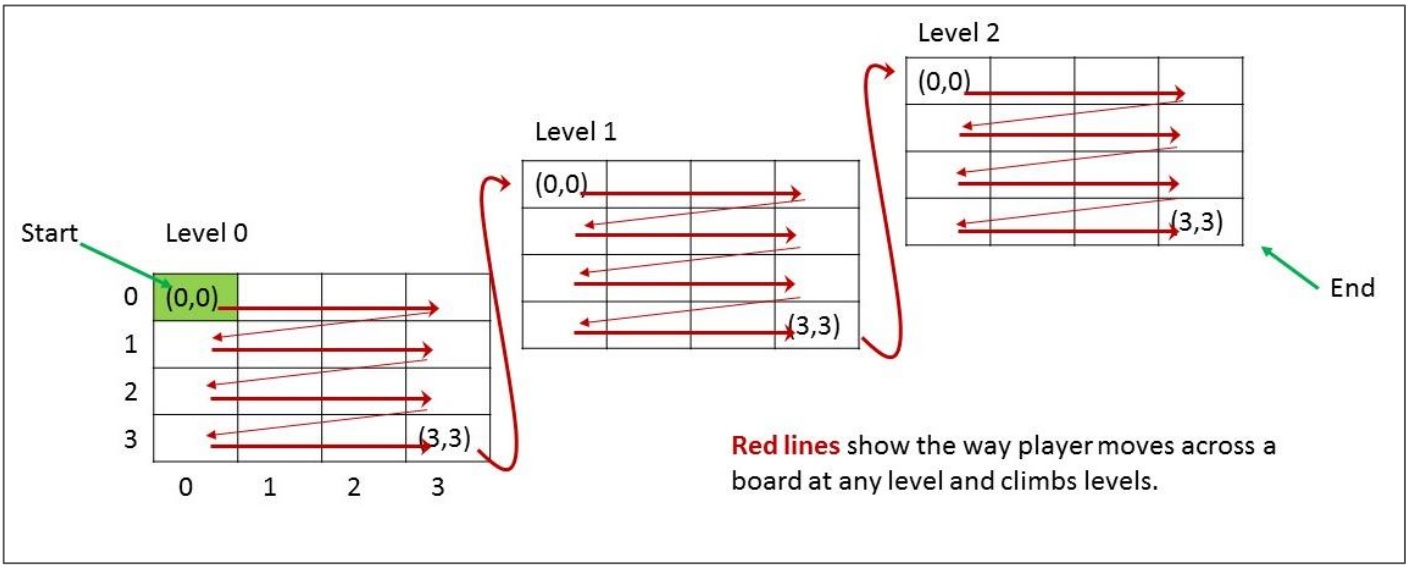

Figure 1- Moving across a board and climbing levels

5. The general **game rules**:
   - Each player tosses the dice and moves the specified number of squares on the board at their current level.
   - If a player reaches the end of a board, s/he continues onto the board on the next level. If the player is already on the last level, then s/he does not move and loses two energy units. (To win a player must land on the last square of the last level.)
   - Each square a player lands on, has energy adjustment units. So a player can gain or lose energy units on some squares. The adjustment units are assigned when the game board is created. (More details about this in the Board Class description below)
   - If at the start of his/her turn a player has zero or less energy units, they cannot move as they are too weak. They toss the dice 3 times in an attempt to boost their energy units. Each time they roll doubles (two 1s, two 2s, two 3s, etc…) they earn 2 energy units. So they could end up with an increase of 0, 2, 4 or 6 units of energy during these 3 rolls.
   - If the square a player (will call player A) lands on is already occupied by the other player (will call this player B), A has 2 options:

- To **forfeit** means A moves to the same square but one level down and lose 2 energy units. However, if A is already at level 0, then A moves to square (0,0).
- To **challenge** the other player for his/her spot. B is not affected.
  - If A wins the challenge, then A takes B's position, and B moves to A's original position. Also since B lost the challenge, B loses half of his/her energy units to A.
  - If A loses the challenge, A stays where s/he is originally and loses half his/her energy units. B is not affected.

- The winner is the one who reaches the last square of the last level first.

Part A: Preparation for the *Nancy's 3D Warrior Game* (7 Pts)

Create the following classes and test them before you tackle Part B of this assignment. You can test the methods in each class by writing drivers and calling each method and making sure they behave as you intended them to.

1. Class *Dice* :

1. A *Dice* object has 2 attributes:
   - **i.** an integer *die1* that stores the role of a 1st die.
   - **ii.** an integer *die2* that stores the role of a 2nd die.
2. A default constructor
3. Accessor methods for the *die1* and *die2* attribute.
4. There are **no** mutator methods
5. *rollDice()* which simulates the rolling of 2 dice and assigns a value to *die1* and *die2*. You can use the random number generator of your choice. Remember a roll is between 1 and 6. This method returns the sum of *die1* and *die2*
6. *isDouble()* which returns the Boolean value true if *die1* is equal to *die2* and the Boolean false otherwise.
7. *toString()* which returns a String containing the values of both attributes in a descriptive message. In my sample output it is of the format *Die1: X Die2: X*. You can use the same format or change it.

2. Class *Board* :

1. A *Board* object has 5 attributes:
   - **i.** A 3D array called board
   - **ii.** A static integer constant MIN_LEVEL which is set to 3
   - **iii.** A static integer constant MIN_SIZE which is set to 3
   - iv. An integer *level* which records how many levels this board has.
   - v. An integer *size* which represents the dimensions (size x size) of the game board at each level.
2. A default constructor which sets the *level* to 3, the *size* to 4 and calls the method *createBoard()* to create *board* using the values of *level* and *size* assigned in the constructor. (See below)
3. A constructor that has 2 integer parameters *l* and *x* which represents the number of levels and the number of rows and columns the boards at each level have. Just like the default

constructor, it calls the method *createBoard()* to create the board but uses the passed values of *l* and *x*. (See below)

4. A private method *createBoard()* which has 2 arguments: level and size. It creates the3D - array board using the passed dimensions. It then initialized each location to the following values.
   If the sum of level, x and y:
   i. is a multiple of 3, assign -3 to the location *board[l][x][y]*
   ii. Is a multiple of 5, assign -2 to the location *board[l][x][y]*
   iii. Is a multiple of 7, assign 2 to the location *board[l][x][y]*
   iv. Otherwise assign 0 to the location *board[l][x][y]*
   v. *Note*: Only one of these can be applied and it is the 1st condition that is satisfied in the order listed above.
5. Accessor methods for *level* and *size*
6. *getEnergyAdj(int l, int x, int y)* which returns the *energy* adjustment value stored in that location in the array *board*.
7. *toString()* which returns a string showing the energy adjustments values for each board at each level. (Refer to sample output)

3. Class *Player* :

1. A *Player* object has 5 attributes:
   i. a String *name* that stores a player`s name
   ii. Four integers *level*, *x*, *y* and *energy* which represent the location of a player and the energy units that the player has.
2. A default constructor – which assigns an empty string ("") to *name*, 10 to *energy* and 0 to *level*, *x*, and *y*.
3. A constructor which takes one *String* parameter representing a player's name. It assigns the value of the passed parameter to *name*, 10 to *energy* and 0 to *level*, *x*, and *y*.
4. A constructor which takes 3 integer parameters, represent a location and assigns these values to the new object, 10 to *energy* and an empty string ("") to *name*.
5. A copy constructor which duplicates the passed *Player* object.
6. Mutator methods for all attributes.
7. Accessor methods for all attributes.
8. *moveTo(Player p)* whose purpose is to move the calling *Player* object to the passed *Player* object's location (the *level*, *x*, and *y* of the calling object are changed to the *level*, *x*, and *y* of the passed object). The name and *energy* level of the calling object are not changed. (Useful when swapping locations).
9. *won(Board b)* a *Boolean* method which return true if the calling object is at the last location of the board and false otherwise (top level and largest x and y possible). See *Board* object for details.
10. *equals(Player p)* which returns *true* if the location of the calling object is the same as the location of the passed object. So just compare the *level*, *x*, and *y* of both objects.
11. *toString()* which returns a String containing the values of all attributes of the calling Player object in a descriptive message.

Be sure to test your 3 classes before starting part B.

Part B: Implementation of the game (15 Pts)

Write the class *LetUsPlay* which is the driver class and where all the action happens. Your two players must be stored in an array of type *Player*. There will be one *Dice* object as both players are using the same dice and one *Board* object which represents the game board. Here are the general steps of the algorithm.

1. Display a welcome banner.
2. Ask the user if they want to use the default board size which has 3 levels with 4x4 boards at each level or if they want to set their own dimensions. The minimum number of levels is 3 and the maximum is 10. The minimum board size is 3x3 and the maximum is 10x10. Be sure if the user decides to set his/her own dimensions that they are within these ranges.
3. Create the game board using either the default constructor or the one where you provide the required size. Display the game board
4. Get the player's names and create the *Player* array.
5. Decide which player goes first by generating a random number (0 or 1) which represents the index of the player that goes 1st.
6. And the game begins …..
    a. If the player whose turn it is has energy units ≤ 0 they toss the dice 3 times. Each time they roll a double, their energy increases by 2 units. If they do not increase their units of energy, play moves to the next player as the current player is too weak to move. Note: After the 3 rolls, player could increase by a maximum of 6 energy units and a minimum of 0 units if they do not roll any doubles.
    b. Player rolls dice. If the roll is doubles (two 1s, two 2s, two 3s, etc…), the player's energy level increases by 2 units.
    c. Calculate the new potential location. Be careful don't change the player's location yet, as this move may take them off the board on the top level, or the resulting location may be occupied by the other player.
    Calculating the new location – Here are a few examples assuming the default game board size (3 levels and 4x4 boards on each level)
        - Say the player rolls a total of 9.How many rows and columns must the player move?
        9/size = is the result that needs to be added to the x coordinate and 9%size is the result that needs to be added to the y coordinate. So if the player is at square (0,0) on level 0, 9/4 = 2 while 9%4 = 1, so new x = 0 + 9/4 = 0 + 2 = 2, and new y = 0 + 9%4 = 0 + 1. So the resulting location is (2,1) on the same level. Both the resulting x and y are < size so we are done.
        - Say the player is at location (3,2) and we need to move 9 squares. Using the same procedure x = 3 + 2 = 5 and y = 0 + 1 = 1. So the resulting square is (5,3). But this is off the board at level 0. So the player needs to climb one level. How do we determine which square in level 1 they land on? The new x is now = 5 % size which in this case is 5%4 = 1. So the resulting location is level 1 position (1,3). Here is a visual to explain this move.

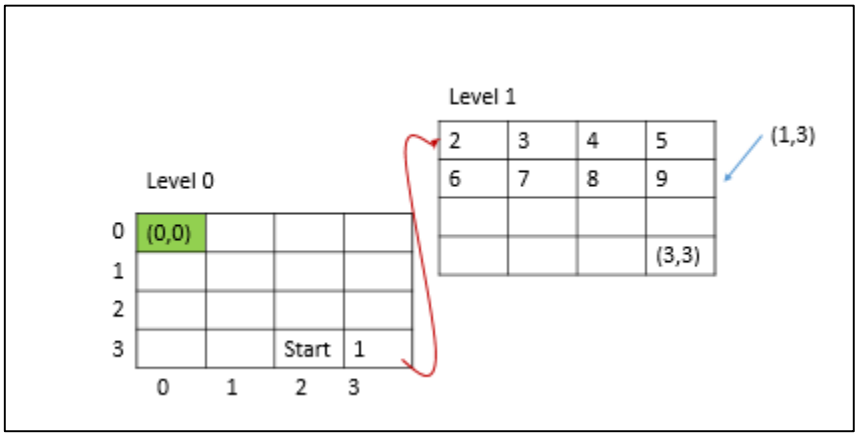

Figure 2- moving from (3,2) level 0 with a roll of 9 to level 1 position (1,3)

- Say the player is as (3,3) and rolls a 9. Using the same procedure 9/4 = 2 while 9%4 = 1, so new x = 3 + 2 = 5, and new y = 3 + 1 = 4. The resulting coordinates are (5,4) on level 0. Both x and y are off of the board.
    1. Because y is off the board, adjust x and y. Start by finding the correct location for the y coordinate. ~~New x = x + 4 / 4 = 4 + 1 = 5~~. New y = 4 % 4 = 0.
    2. After this step, y is within the range. We adjust x (as in the previous example) by adding 4/4 = 1 to 5, so the new x is 6. Just as in the previous example, we go up one level and set x to 6 % 4 = 2. So the new potential location is level 1, (2,0).
- General rule in this order:
    1. Calculate the new x and new y coordinates
    2. If y coordinate is off the board, determine the y coordinate and adjust x accordingly
    3. If x is off the board, determine the resulting x and move up one level.
    4. If the resulting level is >= the number of levels, then the player stays put and loses 2 energy units.

Edited Instructions in pink

d. Before moving to the new location we need to make sure it is not already occupied by the other player.
    - If the spot is free, player moves to that location and adjusts his/her energy levels based on the adjustment unit in the resulting location.

e. If the square a player (will call player A) lands on is already occupied by the other player (will call this player B), A has 2 options:
    - To **forfeit** means A moves to the same square but one level down and loses 2 energy units. However, if A is already at level 0, then A moves to square (0,0).
    - To **challenge** the other player for his/her spot.
        1. The result of the challenge is determined by generating a random number between 0 and 10. If the number is < 6, A lost the challenge.
        2. If A wins the challenge, then A takes B's position, and B moves to A's original position. Also since B lost the challenge, B loses half of his/her energy units to A.

3. If A loses the challenge, A stays where s/he is originally and loses half his/her energy units. B is not affected.

f. The winner is the 1st player to land on the last square of the last level. If the current player is a winner, display a message to that effect and end the game, otherwise it is the next player's turn.

➔ **Restriction**: The only classes you can use for this assignment are the classes 3 classes from Part A, the driver class, `Scanner` and `Random` or `Math`. You are not to use any other classes.

➔ **REMINDER**: A complete sample run can be found in the file *SampleOutput.pdf* you downloaded with this handout. Note that it is missing the display of the deck and discard pile at the end.

Final Comments:

- You can change the messages as long as the same information is there.
- You can use, and in fact are encouraged, to use static methods in your driver class.
- You must have 4 classes, one of which is the driver class. You must have and use the methods given in the specifications of the classes. You can add other methods to the classes.
- It is not advisable to declare a `Scanner` object in more than one class or method. I recommend you create one in the driver class and pass it to any method that requires it.

Submitting Assignment 4

Please check your course Moodle webpage or eConcordia webpage on how to submit the assignment.
Remember to create a .ZIP file and not a .RAR files.
You will have 4 java files to submit.

Evaluation Criteria for Assignment 4 (25 points)

Comments & Programming Style (3 pts.)	
Description of the program (authors, date, purpose)	0.5 Pts.
Description of variables and constants	0.5 Pts.
Description of the algorithm	0.5 Pts.
Use of significant names for identifiers	0.5 Pts.
Indentation and readability	1 Pt.
Part A – Setting up classes for the game (7 pts)	
Class Dice	2 Pts.
Class Board	3 Pts.
Class Player	2 Pts.
Part B – LetUsPlay class /Driver (15 Pts)	
Display welcome banner	0.5 Pts.
Set size of game board/Create corresponding game board	1.5 Pts.
Set up players	2 Pts.
Play Game	
If player's energy <= 0	2 Pts.
If roll doubles (two 1s, two 2s, etc….)	1 Pt.
Calculate new potential location	2.5 Pts.
Two on the same location / Forfeit/Challenge	2.5 Pts.
Move to correct location/Adjust energy units	2 Pts
Game over/ Declare winner	1 Pt.

TOTAL	**25 Pts.**