

## WES-CS GROUP MEETING #10

### Exercise 1: Designing a Class (static vs non-static fields and methods)

For this exercise, we will design a *Student* class. Choose the fields and methods of the class based on the information given below; for each field and each method, choose an appropriate name and type, and decide whether it should be static or not. Then write the class definition.

- Every student has a first name, a last name, a GPA (a number between 0.0 and 4.0), and a 4-digit ID number. The first student's ID number will be 1000, the next student's ID number will be 1001, then 1002, and so on.
- A *Student* object can be created given first and last names only (in which case the GPA is initialized to 0.0), or given first and last names and a GPA. In both cases, the student's ID is initialized appropriately.
- A student's names and ID can be accessed, but cannot be modified. The GPA can be both accessed and modified.
- It should be possible to compare two *Student* objects; one student is considered to be "less than" another student if the first student's name comes first in alphabetical order. Two students with the same name are considered equal.
- It should be possible to find out how many *Student* objects have been created so far.

## Exercise 2: Logical Thinking and Debugging

As you probably know by now, sometimes errors in your program cause exceptions to be thrown when the program is run (e.g., you may get a *NullPointerException* if you forget to initialize some object, or you may get an *ArrayIndexOutOfBoundsException* if you use an index that is too large or too small).

One way to debug your program is to “think backwards” from the point of the exception: what could have happened just before the exception was thrown, and what could have happened just before that, and so on, until you get to the point of the actual mistake in your code.

For this exercise, we will practice thinking backward in the context of a chess game. Keep the following in mind:

- A player is not allowed to move into check. (Among other things, this means that two kings can never be next to each other horizontally, vertically, or diagonally.)
- A pawn normally moves one space forward. However, on its very first move it is allowed to move two spaces forward, and to capture an opponent’s piece it moves one space forward *diagonally*.
- If a pawn arrives at the end of the board, it is *promoted*: replaced by a queen, castle, bishop, or knight.

**Part (a):** Look at the board below, which shows a position in a game of chess. You are told that black moved last. Your job is to figure out what black's last move was, and what was white's move before that. There are two solutions: one works whether the person playing white is sitting at the north or the south end of the board (i.e., whether the white pawn is moving south or moving north). The other solution (which is easier) only works if the person playing white is sitting at the *north* side of the board. Note that the moves are not necessarily good ones, but they are all legal.

**NORTH**

<b>black king</b>		<b>white king</b>					
							<b>white pawn</b>
						<b>white bishop</b>	

**SOUTH**

**Part (b):** Now look at the board below, which shows the final position of a game (the game is over because white has check-mated black).

<b>black king</b>		<b>white king</b>					
				<b>white queen</b>			
		<b>white bishop</b>		<b>white pawn</b>			
			<b>white pawn</b>				
							<b>white pawn</b>
							<b>white bishop</b>

This time, your job is to figure out whether the person playing white was sitting on the north or the south side of the board. To do that, you will need to figure out what white's last move was. There are several possibilities, so to narrow it down, you'll need to figure out what black's last move before that could have been in each case. In all but one case you should find that in fact there would have been *no* possible move for black, and that should let you figure out where the person playing white was sitting.

Good luck!

### Exercise 3: Two-D Arrays

For many board games (e.g., chess, checkers, go) the board can be represented using a 2-dimensional array of integers, where the value in position [j][k] tells what piece (if any) is currently at that position. When deciding whether a player has won the game, it is often useful to look for certain patterns on the board.

These ideas lead to the following (partially specified) class.

```
public class BoardGame {
    /*** fields ***/
    private int[][] board;

    /*** public methods ***/
    public BoardGame() { ... } // constructor
    public boolean hasPattern( int[][] pattern ) {
        // part (b)
    }

    /*** private methods ***/
    private boolean isMatch(int[][] pattern, int row, int col) {
        // part (a)
    }

    private boolean isMatchIgnoreZeros(int[][] pattern,
                                       int row, int col) {
        // part (c)
    }
}
```

**Part(a):** The `isMatch` method of the `BoardGame` class should return `true` iff the given pattern matches the portion of the board that has the given row and column numbers as its upper-left corner.

For example, assume that the board array is as follows:

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

Below are the results of some calls to `isMatch`.

pattern array	row	col	value returned by the call <code>isMatch( pattern, row, col )</code>						
<table border="1"> <tr> <td>1</td> <td>2</td> <td>3</td> </tr> <tr> <td>5</td> <td>6</td> <td>7</td> </tr> </table>	1	2	3	5	6	7	0	0	true
1	2	3							
5	6	7							
<table border="1"> <tr> <td>1</td> <td>2</td> <td>3</td> </tr> <tr> <td>5</td> <td>6</td> <td>7</td> </tr> </table>	1	2	3	5	6	7	0	1	false
1	2	3							
5	6	7							
<table border="1"> <tr> <td>6</td> <td>7</td> </tr> <tr> <td>10</td> <td>11</td> </tr> </table>	6	7	10	11	1	1	true		
6	7								
10	11								
<table border="1"> <tr> <td>6</td> <td>7</td> </tr> <tr> <td>10</td> <td>11</td> </tr> </table>	6	7	10	11	0	1	false		
6	7								
10	11								
<table border="1"> <tr> <td>6</td> <td>7</td> </tr> <tr> <td>10</td> <td>14</td> </tr> </table>	6	7	10	14	1	1	false		
6	7								
10	14								

Write method `isMatch`. Assume that both the `board` field and the `pattern` parameter are non-empty, rectangular arrays.

**Part(b):** The `hasPattern` method of the `BoardGame` class returns `true` iff the given pattern occurs *somewhere* in the board array.

For example, assume again that the board array is as follows:

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

Below are the results of some calls to `hasPattern`.

pattern array	value returned by the call <code>hasPattern( pattern )</code>						
<table border="1"> <tr> <td>1</td> <td>2</td> <td>3</td> </tr> <tr> <td>5</td> <td>6</td> <td>7</td> </tr> </table>	1	2	3	5	6	7	<b>true</b>
1	2	3					
5	6	7					
<table border="1"> <tr> <td>6</td> <td>7</td> </tr> <tr> <td>10</td> <td>11</td> </tr> </table>	6	7	10	11	<b>true</b>		
6	7						
10	11						
<table border="1"> <tr> <td>12</td> </tr> </table>	12	<b>true</b>					
12							
<table border="1"> <tr> <td>6</td> <td>2</td> </tr> <tr> <td>7</td> <td>3</td> </tr> </table>	6	2	7	3	<b>false</b>		
6	2						
7	3						
<table border="1"> <tr> <td>1</td> <td>3</td> </tr> <tr> <td>5</td> <td>7</td> </tr> </table>	1	3	5	7	<b>false</b>		
1	3						
5	7						

Write method `hasPattern`. Assume that both the `board` field and the `pattern` parameter are non-empty, rectangular arrays.

**Part(c):** Sometimes when you look for a pattern you only care about matching some parts of the pattern. For example, to look for a diagonal win for X in tic-tac-toe (three X's in a row on the diagonal, using 1's to represent X's), all you care about is whether there are three 1's in a row on a diagonal-- you don't care what values if any are around them. You'd like to be able to use patterns like these:

1	0	0
0	1	0
0	0	1

0	0	1
0	1	0
1	0	0

where the ones need to be matched, but the zeros in the pattern can correspond to any values in the board array.

Write the `isMatchIgnoreZeros` method so that only non-zero entries in the pattern have to match the corresponding positions on the board.

## Exercise 4: Connect-4

Use the remaining time to play the Connect-4 game. Think about how you might use the `BoardGame` class to program it. You could write the program to allow two people to play against each other (in which case your code would have to decide when someone wins), or you could write the program to allow a person to play against the computer (in which case your code would also have to try to prevent the person from winning by getting four in a row, and it would need to have some strategies for getting four in a row itself).

If you like the idea, maybe we'll spend some time at our next meeting working on a Connect-4 game.