Name: _____

CS159 Midterm #1 Review

1. Choose the best answer for each of the following multiple choice questions.

   (a) What is the effect of declaring a class member to be `static`?

   - ☐ It means that the member cannot be modified.
   - ☐ It means that the member is associated with the class, not with objects of the class.
   - ☐ It means that name of the member must be in all-caps.
   - ☐ It means that the member can only be accessed by classes in the same package.

   (b) How many Strings are created when the following statement executes?

```
String[][] stringArray = new String[3][6];
```

   - ☐ 0
   - ☐ 9
   - ☐ 18
   - ☐ There is no way to know.

   (c) A `PrintWriter` can be used without creating a `FileWriter` object. Why might we create a `FileWriter` as well?

   - ☐ A `PrintWriter` alone can't be used for file output. It is only used for terminal output.
   - ☐ A `FileWriter` can be used to append to an existing file, a `PrintWriter` alone cannot.
   - ☐ The `FileWriter` class includes many convenience methods for formatting output.
   - ☐ A `FileWriter` is unbuffered, so it saves us the bother of calling `close`.

2. Answer each of the following short answer questions. Your answers should be clear and concise. I will not give credit for answers I cannot read or understand.

   (a) Assuming that `numbers` is a properly initialized two-dimensional array of integers, what is the effect of the executing the following code? You may assume that `i` and `j` are in bounds.

```
int[] a;
a = numbers[i];
numbers[i] = numbers[j];
numbers[j] = a;
```

   (b) Briefly describe the difference between "checked" and "unchecked" exceptions. Why are some exceptions checked while others are unchecked?

   (c) Why is it uncommon to use try-catch blocks to handle NullPointerExceptions?

3. This question refers to the `WordSaver` class provided on an attached page.

   The `WordSaver` class represents an unfinished effort to write a command line utility that saves the same word to a file multiple times. When completed, this program will read three values from the terminal: a file name, a word, and a number. It will then write the provided word to the indicated file the indicated number of times. For example, if the arguments were:

   ```
   out.txt hello 3
   ```

   Then the file out.txt would be created, and would contain:

   ```
   hello
   hello
   hello
   ```

   The author has included several print statements throughout the program to help her debug the exception handling code.

   What will be printed to the terminal when each of the following are entered as command line arguments?

   (a) `out.txt hello 3`

   (b) `out.txt hello`

   (c) `unwritablefile.txt hello 3`
   (You may assume that `unwritablefile.txt` is a file that cannot be opened for writing.)

4. The following method converts an array of strings to lower-case:

```
/**
 * Takes an array of words and returns a new array containing lower-case
 * versions of each word.
 *
 * @param words An array of (non-lower-case) words.
 * @return A new array containing lower-case words.
 */
public static String[] makeLower(String[] words)
{
    String[] lower = new String[words.length];

    for (int i = 0; i < words.length; i++)
    {
        lower[i] = words[i].toLowerCase();
    }

    return lower;
}
```

Rewrite this method so that it uses ArrayLists rather than arrays. Your finished method should conform to the Javadoc and method signature below. For full credit, your solution must use an enhanced for loop.

```
/**
 * Take an ArrayList of words and returns a new ArrayList containing lower-case
 * versions of each word.
 *
 * @param words An ArrayList of (non-lower-case) words.
 * @return A new ArrayList containing lower-case words.
 */
public static ArrayList<String> makeLower(ArrayList<String> words)
{
```

5. Traditionally, tic-tac-toe is played on a 3x3 board. The first player to line up three X's or O's horizontally, vertically or diagonally wins. For this question, you will work on the implementation of a generalized version of tic-tac-toe that can be played on a rectangular board of arbitrary size. In this version of the game, the first player to completely fill any row or column with his or her own mark wins. For example, the following board shows a vertical win by X on a 2x4 board:

X	X	O	
O	X		

Any board size is allowed, although the game is not very interesting if the width or height is equal to one. It is not possible to win on a diagonal.

Complete the unfinished methods in the following class according the the Javadoc comments. You are free to add helper methods if you wish. The amount of space provided for each method does *not* necessarily indicate how much code is required.

Your code does not need to handle any error conditions that are not explicitly described in the Javadoc comments. You may assume that the class `InvalidMoveException` is an existing exception class.

```
/**
 * This class represents a generalized tic-tac-toe game. The board is
 * represented as a two dimensional array of strings, where each entry has one
 * of three values: " " - for positions that have not yet been played. "X" - for
 * positions that have been played by X. "O" - for positions that have been
 * played by O.
 */
public class TicTacToe
{
    private String[][] board;
    private int width;
    private int height;

    /**
     * Construct a tic-tac-toe board with the provided dimensions. All positions
     * are initialized to contain a single space (" ", not "" or null).
     *
     * @param width The width of the board
     * @param height The height of the board
     */
    public TicTacToe(int width, int height)
    {









    }
```

```
/**
 * Check to see if a particular board position is playable. A position is
 * playable if it is inside the bounds of the grid and no one has yet played
 * there.
 *
 * @param row the row to check
 * @param col the column to check
 * @return true if the indicated location is playable.
 */
public boolean isPositionPlayable(int row, int col)
{









}

/**
 * Place either an X or an O in the indicated board position.
 *
 * @param row the row to play
 * @param col the column to play
 * @param player either "X" or "O"
 * @throws InvalidMoveException If the indicated board position is not
 *             playable.
 */
public void play(int row, int col, String player) throws InvalidMoveException
{









}
```

```
    /**
     * Check a particular row to see if it contains a win for the indicated
     * player.
     *
     * @param row the row to check
     * @param player either "X" or "O"
     * @return true if every entry in the row matches the player
     */
    private boolean rowWin(int row, String player)
    {









    }

    /**
     * Check to see if the indicated player has a win on any row.
     *
     * @param player either "X" or "O"
     * @return true if any row contains a win for the player
     */
    public boolean anyRowWin(String player)
    {









    }
}
```

```
public class WordSaver
{
    /**
     * The main method passes the command line arguments to saveAll,
     * and prints "File saved!" if the operation was successful.
     */
    public static void main(String[] args)
    {
        try
        {
            System.out.println("A");  // <----- PRINTLINE HERE !
            saveAll(args[0], args[1], args[2]);
            System.out.println("File saved!");
        }
        catch (FileNotFoundException e)
        {
            System.out.println("B");  // <----- PRINTLINE HERE !
        }
        catch (ArrayIndexOutOfBoundsException e)
        {
            System.out.println("C");  // <----- PRINTLINE HERE !
        }
        System.out.println("D");      // <----- PRINTLINE HERE !
    }

    /**
     * Prints the indicated word to the indicated file the correct
     * number of times. Throws a FileNotFoundException if the file
     * can't be opened for writing.
     */
    private static void saveAll(String fileName, String word,
                                String numberString)
            throws FileNotFoundException
    {
        System.out.println("E");      // <----- PRINTLINE HERE !
        int numToPrint = howMany(numberString);
        PrintWriter printer = new PrintWriter(fileName);
        for (int i = 0; i < numToPrint; i++)
            printer.println(word);
        printer.close();
        System.out.println("F");       // <----- PRINTLINE HERE !
    }

    /**
     * Converts the provided string to an integer.
     * Returns 0 if the conversion fails.
     */
    private static int howMany(String numberString)
    {
        try
        {
            Scanner scanner = new Scanner(numberString);
            int val = scanner.nextInt();
            System.out.println("G");  // <----- PRINTLINE HERE !
            return val;
        }
        catch (InputMismatchException e)
        {
            System.out.println("H");  // <----- PRINTLINE HERE !
        }
        return 0;
    }
}
```