Mind the Boggle

Final Programming Project

Final code due: Mon. aft. lab: May 11@11PM; Mon. eve. lab: May 12@5PM; Tues. lab: May 12@11PM

This project asks you to construct a Java program that implements a version of the game of Boggle. The rules for the game of Boggle are reproduced on the next page. If you are unfamiliar with the game, you should probably read these instructions before reading the description of the program.

This assignment differs from our weekly labs in several ways. First, you have more time to work on it. We will devote the next three lab periods to the construction of this program. Second, this lab handout describes what the program should do and provides some tips on programming techniques that should be helpful, but unlike previous lab handouts, it will not present an overview of the internal structure of the program. We will not tell you what classes and methods to write. That part of the program’s design will be your responsibility. Finally, this assignment will be weighted more heavily when determining your final grade.

You may work in pairs on this project, but it is not required. You may choose to work with someone who normally attends a different lab than you attend. In this case, please inform us which of the labs you and your partner will plan to attend during the three weeks you are working on the final project. You should observe the following timeline for the project.

**Week 1** (4/24-25): Your goal for the first lab should be to explore possible approaches for organizing the code for the program into classes and methods. The main product we want you to submit based on this work will be a typed document describing your plans for the structure and implementation of your program. This should include a brief description of each class you plan to define and the methods each class will contain along with an implementation plan for completing the various classes and methods you propose. This document should be placed in the Dropoff folder for the lab you attend 24 hours before the usual deadline for that lab. We will review your design and implementation plans before the next week’s lab to confirm that your ideas are reasonable.

**Week 2** (5/2-5/3): You should aim for this week to have a program that can support solitaire Boggle. The program should let a player select sequences of tiles that form words, but it need not keep track of what the score is. You must give your instructor a brief demonstration of the state of your program by the end of this lab. You should submit whatever code you have completed by the deadline for your lab period during the week.

**Week 3** (5/9-5/10): Your goal should be to have finished writing the code for the complete assignment by the end of this lab, leaving time for debugging between the end of the lab period and the submission deadline. Again, you should give your instructor a brief demonstration of the state of your program before you leave lab. You should submit your complete project in the Dropoff folder for the lab you and your partner have been attending.

You will find a starter file for this lab with the online version of this handout on the labs page of the course website. This starter includes a class called Lexicon that is discussed near the end of the handout. It provides access to a dictionary of valid words.
Figure 4: SOILS

Figure 5: SOAR

RIGHT

Wrong

TYPES OF WORDS ALLOWED: Any word in a dictionary, or any word formed by combining letters in any order, may be considered a word. Words must be at least three letters long and may not exceed six letters in length. Proper nouns and phrasal verbs are also acceptable. Letters may be repeated as many times as necessary.

SCORING AND WINNING: When the timer runs out, everyone must stop writing. Each player then checks their list against the list of words provided, including any words they may have added during the game. The player with the most words on their list wins. In the event of a tie, the player with the highest total score wins.

NO. OF LETTERS: Each player has a total of 10 letters to work with at the beginning of the game. These letters are randomly selected from a pool of letters provided by the game.

POINTS: Each word on the list is assigned a point value based on its length: 1 letter = 1 point, 2 letters = 2 points, 3 letters = 3 points, 4 letters = 4 points, 5 letters = 5 points, 6 letters = 6 points. The player with the highest total score wins.

The winner is 1) the player whose first list has the highest total score, or 2) the first to reach 100 points, or 3) whichever score is reached first.

The game can be played by any number of players, with each player having a set of letters to work with. The game is won by the player with the highest total score at the end of the game.
A sample of the interface your program might present for the game is shown on the right.

We emphasize “might” because we want to encourage you to be creative in your interface design. While your interface must provide functionality similar to our version, you should feel free to use one that is very different from ours.

Most of the program window shown is occupied by a 4x4 grid of buttons labeled with letters. This grid represents the grid of letter cubes presented in the physical game of Boggle. When a person playing the game identifies a group of letters that form a word in the grid of letters, he or she enters the word by clicking the appropriate buttons in the order that the letters appear in the word. The letters of the word that is currently being entered are displayed above the grid. In the example shown, the letters “PEEL” appear above the grid, indicating that the buttons labeled “P”, “E”, “E” and “L” have been clicked.

If you are looking at this handout online, you will see that the letters “P”, “E” and “E” are displayed in green while the letter “L” is displayed in blue. In our program, the letter on the last button clicked is displayed in blue and the letters on the other buttons clicked while entering the current word are displayed in green. If the next button clicked is adjacent to the last button (the one displaying a blue letter) and not already included in the current word (i.e., it is black rather than green), then the letter on that button is added to the current word. The letter that had been displayed in blue becomes green and the letter on the most recently added button becomes blue.

To indicate that a complete word has been entered, the player clicks a second time on the last (blue) letter of the word. The program then checks to make sure the word is at least three letters long, has not previously been entered, and is a valid word (we provide a class called Lexicon, that makes checking valid words trivial). If it passes these tests then the word is added to the collection of words the player has found and the player’s score is updated.

The player can indicate that a mistake has been made and there is no apparent way to complete a new word using the letters that have been selected by clicking any of the green letters. Doing this will reset the colors of all of the selected letters to black, allowing the player to start over by clicking the first letter of a new word.

The words that have already been found in the grid are displayed in a scrollable JTextArea on the right side of our window. The current score is displayed above this JTextArea.

A JProgressBar is displayed at the top of the window to indicate how much time remains in play. The fraction of the progress bar that is “filled” with blue indicates the fraction of game time that has passed. Boggle uses a 3-minute timer.
While we think the progress bar is a nice touch, it is definitely one of the features of our interface that you do not need to mimic exactly. Feel free to replace it with a JLabel that displays the number of seconds remaining to play. If you want to use a JProgressBar, just Google its name and read the online documentation.

The components at the bottom of our program’s window are used to control when games start and whether the game is played in solitaire mode or with another player through the network. If the “START NEW GAME” button is pressed, the program randomizes the letters displayed in the buttons representing the cubes and begins a solitaire game. If the “Find Partner” button is pressed instead, the program attempts to find a possible partner on the network. Once a partner is found, it starts a two-person game.

Two text fields are provided in the program’s interface for pair play. Before pressing the “Find Partner” button, a player should enter their name in the “Your name” field so that it can be displayed on the opponent’s screen. To provide a way to limit who can be chosen as a partner after “Find Partner” is clicked, the program includes a “Partner Group” field. If this field is not empty when the “Find Partner” button is pressed, the program will only pair with another player if that player typed the same group name in the corresponding field in their Boggle program. In particular, while debugging your program, it will be helpful to enter an identifying code in this field to ensure that your program only gets paired with a second copy of itself.

Once pair play begins, each word entered by either player will be shared with the other player’s program through a server we will provide. Your program should use this information to display each player’s current score in the program window. During pair play, your program should provide some mechanism like the “Concede” button shown in the program window to the left, that a player can use to terminate the game early.

Some Important Details

Shaking Boggle Cubes
When your program first starts or whenever you press Start New Game, your code should randomize the letters shown in the grid of buttons representing the Boggle cubes. As explained below, this will not be necessary during pair play. The server used to identify a partner for pair play will send a randomized set of letters to both player’s programs at the start of play.

To randomize the Boggle cubes for solitaire play, you will use a Java library class named Random. This class makes it possible to generate a sequence of (apparently) random numbers. To use this class, you need to include
import java.util.Random;

in your .java file and declare and initialize an instance variable of type Random as shown:

```java
private Random numberChooser = new Random();
```

Instances of Random function like a “wheel of chance” similar to the image shown below. To use a Random object in this way you will invoke its nextInt method.

For example, invoking

```java
numberChooser.nextInt( 12 )
```

returns a random number between 0 and 11 (inclusive). The parameter provided to `nextInt` determines the upper bound of the range of numbers from which the random value will be selected. The lower bound is always 0. If you want a different range, you should include the invocation of `nextInt` in a larger expression. For example, to generate numbers in the same range as the wheel of chance, you could evaluate

```java
1 + numberChooser.nextInt( 12 )
```

Using a Random object to accurately simulate the randomization of the Boggle cubes requires some careful thought. The game of Boggle uses a set of cubes carefully labeled with letters in a way that is intended to yield a nice mix of consonants and vowels when the boggle box is shaken to randomize the letters. To make your game resemble the original, your program should choose the letters displayed based on the letters appearing on the actual Boggle cubes.

To make this a bit easier to do, we provide the following Java initialized array variable declaration:

```java
private final String[][] cubeFaces =
    new String[][]{
    { "T", "Y", "A", "B", "I", "L" },
    { "J", "M", "O", "Qu", "A", "B" },
    { "A", "D", "E", "N", "V", "Z" },
    { "B", "F", "I", "O", "R", "X" },
    { "D", "E", "N", "O", "S", "W" },
    { "D", "K", "N", "O", "U" },
    { "E", "E", "F", "H", "I", "V" },
    { "E", "G", "I", "N", "T", "V" },
    { "E", "G", "K", "L", "U", "Y" },
    { "G", "I", "L", "R", "U", "W" }
    };
```
You should be able to cut and paste this declaration from the online PDF of this handout into one of your Java class files.

This declaration associates the name `cubeSides` with a two dimensional array of `Strings` (most of which contain just a single letter). This array contains 16 rows and 6 columns. Each row describes the 6 letters appearing on the sides of one of the 16 cubes included in the real Boggle game. If `cubeNumber` and `sideNumber` are `int` variables, then

```
cubeSides[ cubeNumber ][ sideNumber ]
```

will equal the letter(s) found on the specified side of the specified cube. For example, `cubeSides[ 4 ][ 3 ] = “L”` because L appears on the third side of the fourth cube and `cubeSides[ 2 ][ 5 ] = “B”`. Using this array, you can pick a random letter for the first button in the Boggle grid by setting `cubeNumber` equal to a random number between 0 and 15, setting `sideNumber` equal to a random number between 0 and 5 and then using

```
cubeSides[ cubeNumber ][ sideNumber ]
```

as the letter for the first button.

Setting the letters for the other 15 buttons is a bit trickier. For the second button, you must pick a random row of the `cubeSides` array that is different from the one you picked for the first button. For the third button, you will need to pick a random row different from those used for both the first and second buttons, and so on.

A clean way to accomplish this is to imitate a bit of the algorithm you implemented to compute the cost of a Huffman encoding. In that algorithm, you first found a minimum value and then exchanged it with the last element in an array to make it easy to later find the minimum value excluding the first minimum. In your Boggle program, each time you pick a random row of the `cubeSides` array, you should exchange the elements of that row with the last row you have not yet used. If you do this, then after you have assigned letters to 15 buttons, the first 15-L rows of `cubeSides` will hold the letters on the cubes that have not yet been used while the last L rows will hold the descriptions of the cubes that have been used to pick letters for the first L buttons.

Note that “J” and “Qu” only appear once on the Boggle cubes and they appear on the same Boggle cube. So, if your program ever generates a game board with two “J”s or two “Qu”s or one of each, you did not implement the randomization process correctly.

**Button Holes**

Each time a player clicks on one of the letter buttons in the Boggle Grid, the program should first determine whether the button is the first button of a new word or not. If it is not the first button, the program should determine:

(a) whether the button clicked is the same as the last letter clicked, or

(b) whether the button is one of the other buttons already used in the current word, and if not
(c) whether the button is adjacent to the last button added to the current word.

One way to answer questions (a) and (b) is to keep a one dimensional array holding all of the JButtons clicked while forming the current word. This array will have to be big enough to hold 16 buttons, but at any point it will only hold as many buttons as have been clicked while forming the current word. You will keep all of the buttons clicked for the current word in the first few elements of the array and you will use a separate int variable to keep track of how many buttons are in the array.

There are two techniques you might use to maintain the information needed to answer question (c) — arrays and inheritance.

Given your experience with arrays in this course, many of you will probably find using an array the most comfortable approach. In this case, you should create a second, two dimensional array of JButtons mirroring the physical layout of the buttons in the game grid. When a button is clicked, your program will search this array for a match to determine the row and column of the button that was clicked. This information will enable you to tell if the button clicked is adjacent to the last button clicked (as long as you also remember the row and column of the last button clicked).

An alternate approach that exploits the power of inheritance is to define a new class named BoggleButton that extends JButton. In the extended class you can add int instance variables to hold the button’s position in the grid (row and column) and methods to use these instance variables to determine if two buttons are adjacent. To use this second approach, there is one new Java programming technique you will need to know. Even if you define a new class

    public class BoggleButton extends JButton { . . . }

the button clicked method must still be declared to expect a JButton as a parameter:

    public void buttonClicked( JButton which ) { . . . }

The internal mechanisms that arrange for buttonClicked to execute when a button is clicked do not know about BoggleButtons, so if you change the parameter type from JButton to BoggleButton, the method will never be executed. Instead, once you are sure that one of the grid buttons has been clicked, you must use a construct called a “type cast” to tell Java to treat the button clicked as a BoggleButton. As a result, the overall structure of your buttonClicked method will look like:

    public void buttonClicked( JButton which ) {
        . . .
        if ( which == startButton ) {
            . . . // code for the start button
        } else { // we know that “which” must be one of the grid buttons
            BoggleButton whichLetter = (BoggleButton) which;
            . . . // code that uses “whichLetter” as the buttons name
    }
The clause "(BoggleButton)" appearing in the assignment to whichLetter is the type cast. It tells Java to treat the value associated with which as a BoggleButton. This makes it possible to associate the button with the whichLetter variable declared to be a BoggleButton.

**BP = The Boggle Pairing Protocol**

We will provide a server to enable your programs to find one another for pair play. The protocol you should use to interact with this server is quite simple.

The server will be listening for connection requests on port 13416 of the machine named lohani.cs.williams.edu.

When your program’s user requests pair play, your program should establish a NetConnection to the server and sends a request of the form

```
PLAY player-name group-name
```

The player-name will be sent to the partner's client and displayed to the opponent. The group-name is optional. If included it tells the server that the client only want to play with partners who provide the same group name. If no group name is included, then play can be initiated with any other client who specifies no group name.

Once a partner is found, the server will send both players a START command consisting of the word START and the opponent's name on one line followed by separate lines containing a randomly chosen list of letters from Boggle cube faces that should be used as the board for play. The letters will be sent one per line with the upper left cube letter first followed by the remaining letters left to right and top to bottom.

If the player using your program enters a valid word, the program should send a WORD command to the server consisting of the word "WORD" followed by a space and the letters of the word.

When one player finds a word and that player's client reports it to the server, the server will forward the WORD command to the other client program.

When play time is over (after 180 seconds), the server will close the connection to the client programs. If either client unexpectedly closes the connection before play time is over (possibly because the client program encountered a program error), the server will close its connection to the other client.

**Recursion Required!**

When a player clicks on the last letter clicked for a second time to indicate that a new word has been completed, your program will need to make sure that this word had not been entered previously. During pair play, your program will also need a list of all the words each player has entered to recompute the players’ scores when a new word is found. For full credit on this assignment, we require that you implement a recursive list of Strings to perform these operations. This class should provide at least one method: contains.
Timing is Everything!
The real Boggle game includes an hourglass (that runs for much less than an hour) used to limit play time. To implement this functionality, your program should use the PaceMaker class of the Squint library. A PaceMaker object acts like a timer that repeatedly notifies your program when a specified time interval elapses. You will want to continuously update a JLabel or JProgressBar to inform your player how much time remains. So, you will want a PaceMaker that notifies your program every second so that it can update this display. When the display’s value indicates that the time has expired during solitaire play, you should end the game by disabling the game buttons. During pair play, the server will notify your program that play should end by closing the NetConnection between your program and the server. You should still include code to continuously display the time remaining, but you should terminate the game only when the server says to do so (even if your timer runs out first).

The PaceMaker class has a constructor that expects two parameters. The first is a number specifying how much time should elapse between the times at which notifications occur. The second parameter is the name of a GUIManager that will perform the necessary action. Thus, to create a PaceMaker that would remind you to update your time display once per second, you might use the construction

```java
    timer = new PaceMaker( 1, this );
```
assuming you have also included a declaration of the form

```java
    private PaceMaker timer;
```

To use a PaceMaker, you define a method named tick within your GUIManager.

```java
    public void tick() {
        . . .  // code to update time remaining
    }
```

The PaceMaker will arrange to execute the code you place within your tick method at the desired frequency. The tick method should not expect any parameters.

A PaceMaker can be turned off by invoking its stop method:

```java
    timer.stop();
```

More details about this class can be found in the online Squint documentation available in the Documents section of the course web site.

Checking Valid Words.
The Lexicon class provides a way to check whether words that are entered are valid. The easiest way to use Lexicon is to create an instance variable of type Lexicon (say lexicon) and then use the contains method to check whether some variable word of type String is valid. Said differently, if word is valid then lexicon.contains(word) will return true. Otherwise it will return false. The Lexicon class provides a second method named containsPrefix that is not needed for the basic assignment, but can be very handy if you want to add the ability to display a list of all words that appear in the current grid at the end of play.
Submitting Your Work
You should submit both your design and your final code electronically. You can find instructions describing how to submit your program on the “Labs” page of our web site at

http://www.cs.williams.edu/~cs134/Labs.html