Take My Word

Final Programming Project

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

This assignment asks you to construct a Java program that implements a word game we will call “Take My Word” (despite its obvious similarity to an existing game known as “Letter by Letter”). The rules for the game are described below.

This assignment differs from our weekly labs in several way. 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 groups of up to three students on this project. You may choose to work with students who normally attend different labs than you attend. In this case, please inform us which of the labs you will plan to attend during the three weeks you are working on the final project. You should follow 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 most important methods and instance variables each class will contain. It should conclude 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** (5/2-3): You should aim to have a program that at least supports a solitaire version of the game by the end of this week’s lab. The program should let a player place word tiles and select sequences of tiles that form words, but it need not keep track of which player is playing or 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** (5/9-10): 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 must 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(s)) 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.
Rules 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 the same functionality as our version, you should feel free to use one that is very different from ours.

The game is a two player game. The players are only identified by their colors: red and green. Most of the program window shown is occupied by a 7x7 grid of buttons. At the bottom of the window a “tile rack” of 8 buttons displays letters of the alphabet that the player can place in the 7x7 grid to form words. Between the grid and tile rack, the program displays buttons the user can click to complete or retract a move and instructions on what action the player needs to perform. The color of the instructions displayed in this region indicates which player should be making the next move.

A player starts a move by clicking to select a letter in the tile rack to be moved to the board. Once a letter is selected, the program will remove it from the rack and update the instructions to the player as shown on the left. Next, the player selects a position for the letter in the 7x7 grid. The position in which the new letter is placed must either be at the bottom of the grid or directly above another letter already placed in the grid. For the first move, therefore, the player must place a letter somewhere in the bottom row of the grid as shown on the right.

This process is repeated until all the desired letters needed to form the word the player has in mind have been placed on the grid. Each letter used to form a word must be adjacent to the previous letter of the word, but they do not have to fall in a straight line. For example, given the letters shown in the example game board, the player might continue by placing the next letter to the right of C, but then move back over the
C to add the remaining letters as shown on the left. If during this process, the player has a change of heart and decides to use a different word, clicking the “Recall Letters” buttons moves all of the letters placed in the current move back to the tile rack.

Once all the desired letters have been placed, the player indicates the sequence of letters that form the desired word by clicking on them in order. As the user clicks, the program disables the letters that have been used and displays the word formed so far as shown on the right. If the user makes a mistake or thinks of a different move during this process, clicking the “Cancel word” button will undo the letter selections and bring the game back to a state where the user can add more letters or click on “Recall Letters” to start from scratch.

Once all of the letters of the word have been selected. The player clicks “Play word” to complete the move as shown on the left. At this point, the program should display all of the letters used to form the word in the player’s color (red or green). A player’s score is equal to the number of letters displayed in the player’s color.

The program then replaces the letters used from the tile rack during the move with new random letters and changes the color used to display the prompts below the letter grid after a move is complete to signal that the other player should begin.
In subsequent moves, each player must add at least one letter to the grid and must use all the letters added in the word formed, but can also use any letters already in the grid. For example, after the first player entered “CHUG” in our example, the other player might add the letters E and A and use the word EACH as shown on the right. By using a letter previously displayed in the other player’s color, a player may switch the letter’s color, simultaneously increasing the player’s score while decreasing the opponent’s score.

Play continues in this manner until the grid is full or neither player can find a move. For example, the game from which we have shown the first two moves, might continue with the players entering the series of words CHUG, EACH, CHANGE, OUCH, EACH, YOUNG, RANGE, CARING, AGAIN, NATION and NOTION resulting in the game board shown on the left. Can you see a good next move for the green player?

**Networked Play**

In addition to having the ability to let two players sharing a single machine play the game, your program should allow play between two players running your program on separate computers. In fact, since we will specify the protocol to use for networked play, your program should allow play between someone using your program and another player using any other student’s version of this program.

A sample interface for a version of the program supporting networked play is shown on the right. The controls for network play are all shown below the tile rack. If two people want to play using a single copy of your program on one computer, they can ignore these controls and start a game by selecting a letter from the tile rack to place on the game board. If instead, a player want to play with someone using a different computer, the player will start by filling in the name and group fields and then pressing the “Find Partner” button.
We will provide a server to enable your programs to find one another for pair play. The server will be listening for connection requests on port 13426 of the machine named lohani.cs.williams.edu. When someone presses “Find Partner” your program should send a request of the form

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

to our server.

The group name is optional. If included it tells the server that the client only wants to play with partners who provide the same group name. If no group name is included, then our server can pair your program with any other program that sent a PLAY request with no group name.

After sending the PLAY command, your program should disable or at least ignore all of the buttons in its interface except that the “Find Partner” button should be changed to display “Disconnect” and if this button is pressed, your program should close the connection to the server to indicate that the partner search should be aborted.

As long as your player does not abort, once a partner is found, the server will send one partner a STARTNOW command and the other a STARTSOON command. Both commands will contain the command name, a blank, and then the player name provided in the partner’s PLAY command.

If your program receives a STARTNOW, it should enable its interface to allow its player to complete a move. Once a move is completed, your program should send three commands to the server as described below. Then, your program should disable or ignore all of its interface buttons except the “Find Partner/Disconnect” button and the “Send” button until it receives information from the other player’s program indicating that the other player has completed a move.

When either player completes a move, the three commands that should be sent to the server are:

1. a WORD command describing the word just formed to complete a move,
2. a BOARD command describing the updated contents of the game board, and
3. a TILERACK command describing the updated contents of the tile rack.

Each of these commands will consist of a single line of information as described below. The server will forward each command it receives to the opponent’s program.

The WORD command should consist of the letters WORD followed by a space and the letters of the word just formed. If a player chooses to pass on a turn, the program should send a WORD command containing no word followed by appropriate BOARD and TILERACK commands.

The BOARD command should consist of the letters BOARD followed by a space and a string of 49 letters describing the contents of the game board. The first 7 letters in this string will describe the bottom row of the game board, the next seven the row one above the bottom and the last 7 the top row. Each row’s contents will be described from left to right. Each letter in the string will describe the contents and ownership of the corresponding cell in the game board. If the cell on the game board is still empty, a single space (“ ”) should appear in the corresponding spot in the
string provided in the BOARD command. If the cell contains a letter currently owned by the sender of the BOARD command, then that cell should be described by an upper-case copy of the letter. If the cell contains a letter currently owned by the opponent of the sender, that cell should be described by a lower-case copy of the letter found in the cell.

Finally, the TILERACK command should consist of the word TILERACK followed by a space and the 8 letters that now appear in the tile rack. Once a TILERACK command has been received, your program should assume the other player’s move is complete and allow its player to take a turn.

There is one last command included in the protocol for this game. A command starting with the word MESSAGE can be used to send taunting messages from one player to another. Our program sends such a message containing any text typed into the text field to the right of the “Send” button when this button is pressed. Incoming messages are displayed below the text field.

Some Helpful Details

Checking Valid Words.
Your program should check that the sequences of letters each player forms during play are actually words before accepting them. To make this possible, we will provide a starter folder containing a class called Lexicon.

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.

Acting Randomly
When your program first starts and after each move, your code needs to pick letters to add to the tile rack. To do this, you will use a Java library class we have not previously presented, the Random class. 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:

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

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

$$1 + \text{numberChooser.nextInt( 12 )}$$

A Java Random object is equally likely to choose any of the numbers in the range specified when
you invoke nextInt. Unfortunately, for good game play, you do not want your program to be
equally likely to choose any of the 26 letters of the alphabet when you pick new random letters
for the tile rack. Instead, you want the distribution of “random” letters to reflect the distribution
with which letters are actually used in English. As an approximation to this ideal, you should
make the distribution of letters your program uses similar to the distribution of letters in a set of
the tiles that come with the game Scrabble.

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

```java
private final int [] scrabbleFreq = { 9, 2, 2, 4, 12, 2, 3, 2,
  9, 1, 1, 4, 2, 6, 8, 2,
  1, 6, 4, 6, 4, 2, 2, 1,
  2, 1 };
```

The values in this array describe the frequencies with which the various letters of the alphabet
appear in the tiles used to play Scrabble. That is, there are 9 As, 2 Bs, 2 Cs, 4 Ds, 12 Es, and so
on. You should be able to cut and paste this declaration from the online PDF of this handout into
one of your Java class files.

Given this array, you can create an array of Strings containing one letter for each tile in a set of
Scrabble tiles (the sum of the numbers in the array is 98 rather than 100 since we left out the two
blank tiles). Then, to pick a letter, use random to pick a number between 0 and 97 and return the
letter at the corresponding position of your array.

**Button Holes**

Each time the person running your program clicks on one of the JButtons representing tiles on
the game board or in the tile rack, your program will need to figure out which button was
pressed. There are two techniques you might use to determine such a button’s position in the
game board or tile rack --- arrays and inheritance. We want to encourage you to use inheritance.

What we have in mind is to define a new class named LetterButton that extends JButton. In
the extended class you can add int instance variables to hold the button’s position in the game
board (row and column) or tile rack and methods that use these instance variables to where a
button is on the grid or to determine if two buttons are adjacent. You can also add an instance
variable to keep track of which player owns a cell in the game board or any other necessary
information.

To use this approach, there is one new Java programming technique you will need to know. Even
if you define a new class
public class LetterButton 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 LetterButtons, so if you change the parameter type from JButton to LetterButton, 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 LetterButton. As a result, the overall structure of your buttonClicked method will look like:

public void buttonClicked( JButton which ) {
    . . .
    LetterButton whichLetter = (LetterButton) which;

    . . .   // code that uses “whichLetter” as the buttons name
}

The clause “(LetterButton)” appearing in the assignment to whichLetter is the type cast. It tells Java to treat the value associated with which as a LetterButton. This makes it possible to associate the button with the whichLetter variable declared to be a LetterButton.

**Recursion Required!**

As a player clicks letters in the game board to form a word, your program will need to keep track of the sequence of letters used, and their positions on the board. While a String of letters would be sufficient if all you needed to do was keep track of the word itself, you need to know the actual positions of the tiles selected so that you can switch the colors of the appropriate letters on the game board when the word is completed. If you use a LetterButton class as we suggest above, you can accomplish this by keeping a sequence of LetterButtons rather than a sequence of Strings.

This sequence could be stored as an array, but for full credit, we ask you to use a recursive list to hold this information. That is, you will define a class of LetterButton lists. It will probably be handy if this class contains a toString method that returns the word formed by the sequence of buttons and another method to set the colors of all the buttons in the list to a color/player specified by a parameter.

**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