Unit 6 Project: Wordle!

This project is designed to help you practice working with strings, arrays, and objects in the context of an engaging application: the Wordle game initially developed by Josh Wardle, now available on the *New York Times* website. Given Wordle’s enormous popularity, we thought it would be fun to give you the chance to implement the game. If you are unfamiliar with the game, you may want to play it. If you Google “wordle”, it’ll show up.

We’re treating this as part of the Arrays unit, but it’s really more of a “midterm” project. You’ll need all of the tools you’ve learned this year!

***BE ADVISED!!!!!!***

------------------------------------------------------------------------------------------------------------

Please note that we are well aware that code and implementations for Wordle exist online. This assignment has been tailored specifically for this class *this year* and submissions will be subject to plagiarism detection using **very fancy software written by very fancy people at Stanford**. Not only will your assignment be compared against online solutions, it will be compared against your classmates. This very fancy software is SO FANCY that even changing variable names and moving chunks of code around won’t fool it as the code is compared at the *abstract syntax tree* (AST) level.

Sharing/copying code from your classmates… cheating!

Using AI-generated code… cheating!

Using websites such as StackOverflow for code segments… not *exactly* cheating (if you cite your sources with a comment… but tread… very… carefully…

Frankly, *you’re probably not as good at cheating as you think you are*, and 90% of the time it’s obvious just from reading your code. Just do the work. It’s fun!

------------------------------------------------------------------------------------------------------------

***YOU HAVE BEEN ADVISED!***

# The Starter Code

The good news is that you don’t have to implement the Wordle project entirely from scratch. The starter code project comes with 4 files, three of which you will not need to update, and are hidden from you in the folders:

* WordleDictionary.java - contains an array called FIVE\_LETTER\_WORDS, from which a solution is picked, and all guesses are compared against to see if they are valid. You do not need to edit this file, but you may want to look at it.
* WordleGWindow.java - contains the code to build the interactive graphical window. You do not need to edit or review this file.
* WordleEventListener.java - contains the code to tie keyboard and mouse events to WordleGWindow. You do not need to edit or review this file.
* Wordle.java - this is where you are going to write your code.

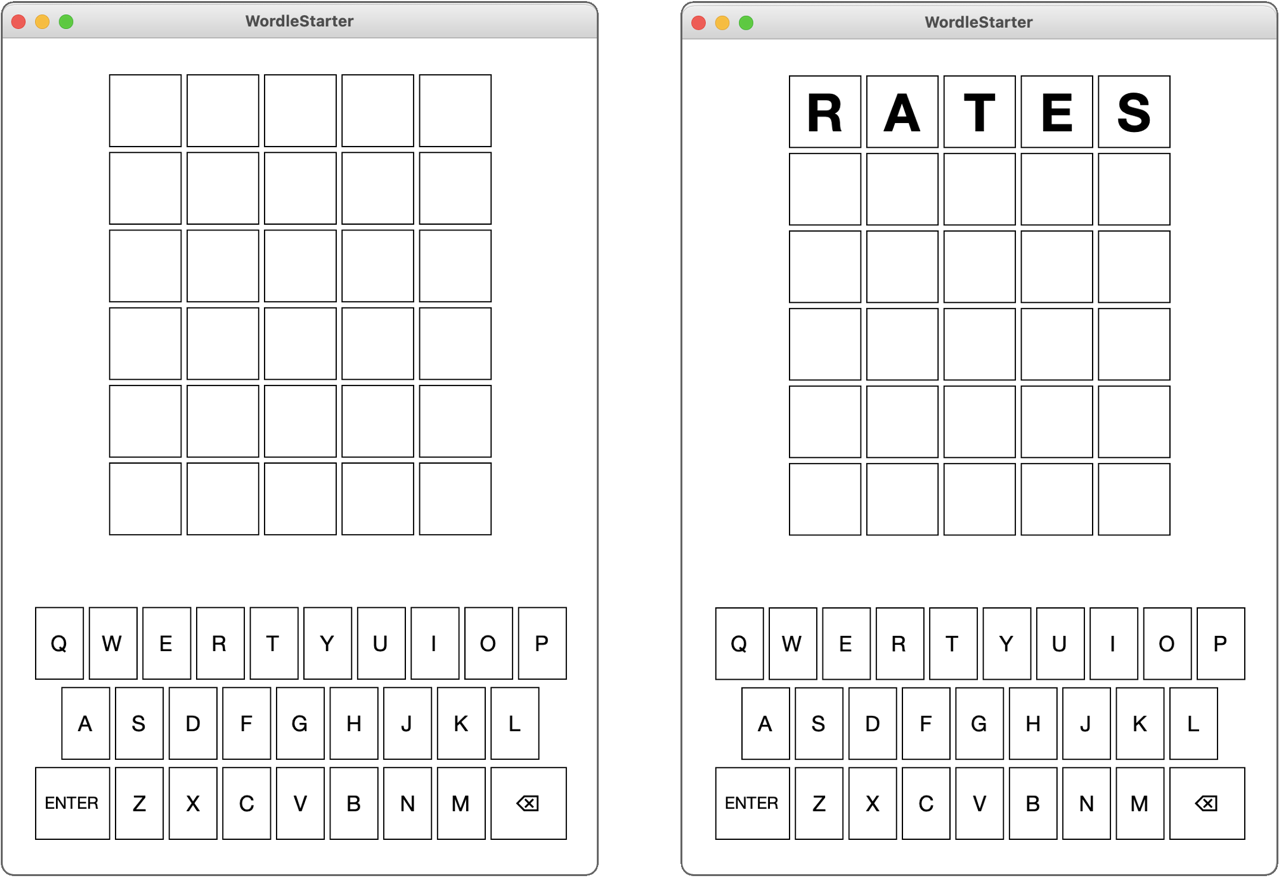
As the starter file shows, you can import these other useful classes into your program using the lines:

import edu.willamette.cs1.wordle.WordleDictionary; import edu.willamette.cs1.wordle.WordleGWindow;

When you run the project for the first time, a lot of the code is already running because we’ve implemented the graphics for you. Running the starter program creates a window, draws the letter boxes, and creates the keyboard at the bottom of the window. You can even type in letters either by hitting keys on the keyboard or clicking the keys on the screen, just as you can when you are playing the online version. Figure 2, for example, shows both the initial screen and the screen you get after typing in the five letters in the useful starting word RATES, which includes five of the most common letters.

Unfortunately, that’s all the program does at this point. It doesn’t actually let you play the Wordle game. That’s your job! YAY! But first, it is worth spending a bit of time reviewing the rules for Wordle, in case you’ve somehow managed to miss this craze.

**Figure 2. Running the starter program**



# Playing Wordle

The object of the Wordle puzzle is to figure out the hidden word using no more than six guesses. When you type in a word and then hit the RETURN or ENTER key, the website gives you information about how close your guess is by coloring the background of the letters. For every letter in your guess that is in its correct position, Wordle colors the background a light shade of green. For every letter that appears in the word but is not in the correct position, Wordle colors the background a brownish yellow. All letters in the guess that don’t appear in the word are colored a medium gray. These colors are defined in the WordleGWindow class, which means that you can refer to them in your program using the following constants:

WordleGWindow.CORRECT\_COLOR WordleGWindow.PRESENT\_COLOR WordleGWindow.MISSING\_COLOR

For example, suppose that the hidden word for the day was RELIC, and your first guess was RATES as in the Figure 2 example. The R is in the correct position, and the word contains an E, but not in the position you guessed. The hidden word does not contain any of the letters T, E, and S. Wordle reports that information by changing the background colors of the squares like this:



Even though you know the position of the R, it doesn’t make sense to guess more words beginning with R at this point because doing so gives you no new information. Suppose that you tried guessing the word LINGO, which contains five new letters, two of which appear in the word, but none of which are correctly positioned. Wordle responds by coloring the letter squares in your second guess as follows:



Putting these two clues together means that you know that the word begins with an R, contains the letters E, L, and I in some order other than the one you guessed, and that the letters A, T, S, N, G, and O do not appear anywhere in the word. These answers give you an enormous amount of information. If you think carefully about it, you might find the word RELIC, which is in fact the only English word that meets these conditions:



Done in three!

It is worth noting a few other rules and special cases. The hidden word and each of your guesses must be a real English word that is five letters long. The WordleDictionary class included with the starter package exports the constant FIVE\_LETTER\_WORDS as:

public static final String[] FIVE\_LETTER\_WORDS = [ "aahed", "aalii",. . . , "zoril", "zowie"

];

…where the three dots are placeholders for more than 5000 other five-letter words. If you guess a word that is not in the word list, Wordle displays a message to that effect, at which point you can delete the letters you’ve entered and try again. Another rule is that you only get six guesses. If all the letters don’t match by then, Wordle gives up on you and tells you what the hidden word was.

The most interesting special cases arise when the hidden word and the guesses contain multiple copies of the same letter. Suppose, for example, that the hidden word is GLASS and you for some reason guess SASSY. Wordle responds with the following colors:



The green S shows that there is an S in the fourth position, and the yellow S shows that a second S appears somewhere else in the hidden word. The S in the middle of SASSY, however, remains gray because the hidden word does not contain three instances of the letter S.

**The WordleGWindow class**

Even though you don’t have to make any changes to it or understand the details of its operation, you need to know what capabilities the WordleGWindow class has on offer so that you can use those facilities in your code. The most important thing to know is that this library module exports a class called WordleGWindow, which implements all the graphical capabilities. The methods exported by the WordleGWindow class are outlined in Figure 3 at the top of the next page. The right column of the table gives only a brief description of what these methods do. More complete descriptions appear later in this handout in the description of the milestone that requires them.

# Planning the implementation as a sequence of milestones

Whenever you are working on a programming project of any significant size, you should never try to get the entire project running all at once. A much more effective strategy is to define a series of milestones that allow you to complete the project in stages. Ideally, each milestone you choose should be a program that you can test and debug independently. Similarly, it often makes sense to defer more the more complex aspects of a project until after you have gotten the basic foundation working. The next few sections outline four milestones for the Wordle project that walk you through different stages of the implementation. You should get each one working before moving on to the next one.

# Milestone #1: Explore the code, and pick a random word

1. Run the game, and see how it responds to keypresses, and mouse clicks.
2. Read over Wordle.java in its entirety. The code is extensively commented and the //TODO comments specify where you will enter code
3. In public void run(), read the TODO comment. You will pick a random word from the word list and store it in the solution instance variable. java.util.Random is already imported for you.

# Milestone #2: Check whether the letters entered by the user form a word

Although the starter program lets the user type letters into the Wordle game, hitting the RETURN key simply generates a message saying you pressed enter. The linkage between the WordleGWindow class and the main program occurs through a ***callback function****,* which is a function supplied by a client to a library, which can later call that function to execute that operation on the client’s behalf. In this case, the main program makes the following call to register its interest in being notified whenever the user hits the RETURN key or clicks the ENTER button:

gw.addEnterListener((s) -> enterAction(s));

The argument in this call is an example of a Java ***arrow function****,* which is a convenient bit of syntax for a function definition in which the argument list appears to the left of the two-character arrow (->) and the body of the function appears to the right. Note that no type declarations are required here. The Java compiler simply looks at the definition of addEnterListener to determine the type of function it expects. In this case, that definition tells the compiler that addEnterListener requires a function that takes a string and returns no value. The argument (s) -> enterAction(s) matches that definition and produces a function that takes a string as its argument and then calls the enterAction method, passing along the string s. The effect of this call—mysterious as its syntax may seem at first—is to trigger a call to enterAction in Wordle.java whenever the user hits RETURN or clicks ENTER, passing the five letters on the current row as a string. For Milestone #2, your job is to write code enterAction that checks to see whether the word passed from WordleGWindow is a legitimate English word. If it isn’t, your implementation of enterAction should call the showMessage method with the string "Not in word list", which is what the *Times* website says. If it is a word, you should display some more positive message that shows that you got this milestone running. You should use the following methods in the table on the next page to call on the “gw” object that stores the current game window. You may or may not use all of them, it will depend on your implementation of your game.

Write your code where you see //TODO - Milestone 2. You may find the string .toUpperCase() and .toLowerCase() functions useful. The “guess” parameter is passed in in UPPERCASE, but the WordleDictionary.FIVE\_LETTER\_WORDS list is in all lowercase. At some point you will need to do the necessary case conversions.

**Figure 3. Methods exported by WordleGWindow class**

|  |  |
| --- | --- |
| new WordleGWindow() | Creates and displays the graphics window. |
| setSquareLetter(*row*, *col*, *letter*) | Sets the letter in the specified row and column. |
| getSquareLetter(*row*, *col*) | Returns the letter in the specified row and column. |
| setSquareColor(*row*, *col*, *color*) | Sets the color of the specified square. |
| getSquareColor(*row*, *col*) | Returns the color of the specified square. |
| setCurrentRow(*row*) | Sets the row in which typed characters appear. |
| getCurrentRow() | Returns the current row. |
| setKeyColor(*letter*, *color*) | Sets the color of the specified key letter. |
| getKeyColor(*letter*) | Returns the color of the specified key letter. |
| showMessage(*msg*) | Shows a message below the squares. |

# Milestone #3: Color the boxes

Read the comments for “TODO - MILESTONE 3” in the code.

For this milestone, you need to add code to enterAction that, after checking to make sure it is a legal word, goes through and colors the boxes to show the user which letters in the guess match the word. The method you need to accomplish this task is

gw.setSquareColor(*row*, *col*, *color*)

The row and column arguments are the same as the ones you used to set or get the letters from the boxes, and *color* is the color you want to use for the background, which will typically be one of the constants CORRECT\_COLOR, PRESENT\_COLOR, and MISSING\_COLOR imported from WordleGWindow, which have the following values, each of which is defined in terms of a six-digit hexadecimal number that gives the red, green, and blue intensities.

The hard part of this milestone is figuring out how to color the squares, which is not as easy as it might at first appear, particularly when the hidden word contains multiple copies of the same letter. You need to keep track of which letters in the guess have been used and cross them off as you assign colors. You also need to find the correct colors first so that you don’t end up coloring a letter yellow that will later turn out to be in the correct position.

**Hints:** Start by seeing if they’ve gotten the whole word correct. That’s easy. If not, check each character to see if it is in the correct position and color it green if it is. That is also fairly easy, since you’re comparing the character array indexes one-to-one. The challenging scenario is finding correct characters in incorrect positions. This becomes ESPECIALLY challenging if the word has duplicate characters in it. Make sure you test your code with a variety of words and combinations. There are many different ways to solve this problem. You may need to loop through the word multiple times, you may need to loop backwards, you may need to define extra variables to check each position.

Whenever the user enters a guess that appears in the word list, your program must do a few things. First, it must check to see whether the user has correctly guessed all five letters, in case you want to have your program display some proper congratulatory message. If not, your program must move on to the next row. This information is maintained inside the WordleGWindow class (which needs this information to know where typed letters should appear) using the setCurrentRow and getCurrentRow methods.

This is also where you’ll want to win/lose the game. Use gw.ShowMessage(“message”) to show an appropriate win/lose message and tell the player to press enter to play again. The logic to reset the game has already been coded for you. You’ll just need to set gameOver = true;. The player wins if they get the whole word in 6 guesses or fewer; otherwise they lose.

**So how do we keep track of all of this?** My suggestion is to utilize some arrays. Perhaps some arrays that have 26 elements… one representing each character of the alphabet in order to keep track of how many times a letter has been guessed in a turn, and how many of a particular character exist in the solution. You may at multiple points be tempted to write a 26 clause if statement to check against every single letter (**don’t do this**. *This wildly inelegant solution will not earn you full marks*.)

You should make use of the Java character primitive type. The char type represents a single character:

char value = ‘s’;

Note the single quotes. The interesting thing about Java is that char values can be represented as integers. You will need to consider this when determining what indexes to update in your arrays. How does your program know that the character ‘d’ should go in arr[3]. If you consult the ASCII table below that shows the Decimal representation of each Character, you can see that for a lowercase character, we can get its alphabetical position (starting at zero) by subtracting 97 from the decimal character value.

See the table on the next page. If you don’t want to mess with Characters, stick with Strings, and use the comparison operations you already know.

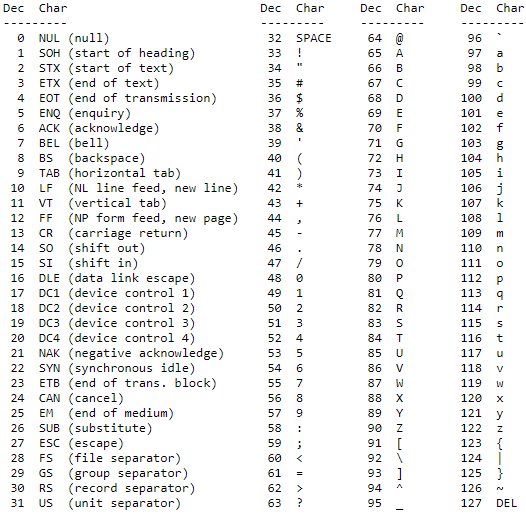
As you are testing your solution, make sure to choose a variety of guess words/solution words. Things get tricky with repeated letters, and repeated letters that come before/after the word in the solution. For example, if the solution is “CLASS” and the player guesses “ALARM” the game should look like this:



There is only one “A” in the solution. An incorrect (albeit easier) behavior would color the first “A” yellow since there technically is an “A” in the solution somewhere else.

**To be clear on the game behavior**:

1. Letters in the right place take precedence and should be considered/colored first
2. Letters in the wrong place should only be colored IF that letter is remaining in the solution after all other letters (right place and wrong place) have been considered starting from left to right
3. Letters should be colored gray otherwise



So for example, the character ‘d’ is decimal 100, 100 - 96 is 4, which makes sense because ‘d’ is the 4th letter of the alphabet. But remember we’re going to be using a zero-indexed array to represent alphabet positions, so we’ll really need to subtract 97 to adjust for that. You can use a casting function to do this like so:

char myChar = ‘d’;  
int alphaPositionIndex = (int) char - 97;

You may also find the .charAt(index) function useful. We know how to pull out an individual letter from a string by using substring, but charAt makes it a little simpler and actually returns a character type:

String myString = “hello”;  
// letterString contains the value “e”  
String letterString = myString.subString(1,2);

// letterChar contains the value ‘e’ (single quotes)  
char letterChar = myString.charAt(1);

# Milestone #4: Color the keys

Your next milestone implements a very helpful feature from the *New York Times* website in which it updates the colors of the keys on the virtual keyboard, making it easy to see what letters you’ve already positioned, found, or determined not to be there. The WordleGWindow class exports the methods setKeyColor and getKeyColor to accomplish this task. These methods use the same string codes as the corresponding methods for squares.

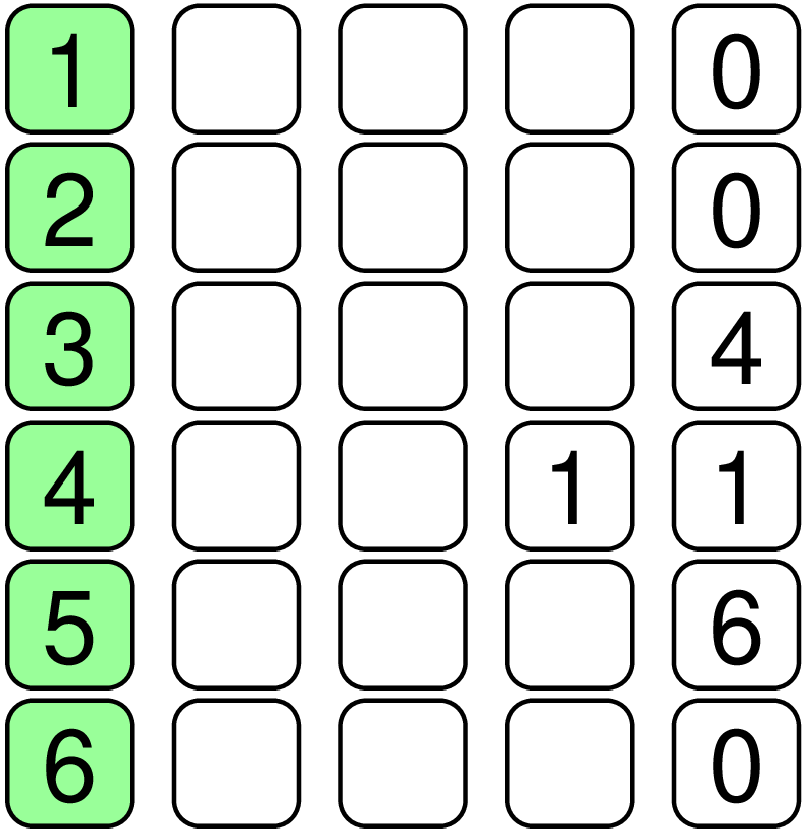
In solving this milestone, it is important to remember that once you have set the color of a key, **it won’t change back**. If, for example, you’ve colored the S key green, it will never get set to yellow or gray even though you may end up using those colors for squares that contain an S. Remember the keys are represented by **uppercase letters**.

**Thoughts to keep in mind**

* As with any large program, it is essential to get each milestone working before moving on to the next. It almost never works to write a large program all at once without testing the pieces as you go.
* You have to remember that uppercase and lowercase letters are different in Java. The letters displayed in the window are all uppercase but the FIVE\_LETTER\_WORDS constant is an array of lowercase words. At some point, your code will have to apply the necessary case conversions.
* Your solution needs to include arrays in a useful fashion. Points will be deducted for a 26-way if/switch statement. Aside from that as long as your game functions correctly, you can write the code however you want.

# Milestone 5:

* *Keep score:.* The *New York Times* Wordle site keeps track of the number of games you’ve played and presents a graph of the number of guesses you needed. Although you could print this information to the console, you could also display the counts in the Wordle grid, so that each row shows the number of times you needed that many guesses. Thus, if you had solved four Wordle problems in three guesses, eleven in four guesses, and six in five guesses, your Wordle program might show the following display at the end. You would only need to keep track of the scores of games played since the application was opened:



**Extra Credit (5 points)**:

Using the File class, persist the scoreboard to disk. In other words, it should keep track of the scores in between sessions of the game being stopped and started.