CSCI165 Computer Science II
Lab Assignment

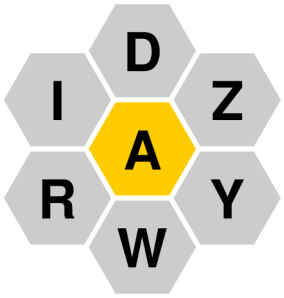
Objectives

- Practice working with strings, array lists, and text files in Java.
- Practice with looping through files and sequences
- Practice performing some basic comparisons on strings and characters
- Work with a cool puzzle game that implements a graphical user interface (GUI) using Java 2D Graphics

The Spelling Bee puzzle

One of the most popular features in the *New York Times* (and one that produces a surprisingly large revenue stream for the paper) is the Spelling Bee, which appears each day on the web at https://www.nytimes.com/puzzles/spelling-bee.

Each Spelling Bee puzzle consists of seven hexagons arranged in a small beehive-like shape. For example, the Spelling Bee puzzle from September 4, 2019 looks like this:



Your task in the puzzle is to find as many words in this layout as you can, subject to the following rules:

- **Rule One:** ***Each word must be at least four letters long***, which means that the word ADZ (an axe-like tool in which the blade is perpendicular to the handle) is too short to be acceptable.
- **Rule Two:** ***Each word cannot contain any letters other than the seven letters in the layout***, although it is legal to use the same letter more than once. For example, you can form WAYWARD from the letters in the grid by using both the W and the A twice.
- **Rule Three:** ***Each word must contain the center letter at least once***, which rules out the word WIRY.
- **Rule Four:** ***Each word must be a legal English word***. The *New York Times* uses a dictionary of "common" words that is more restrictive than a standard dictionary. Unfortunately, the *New York Times* does not publish a list of the words it considers legal (and also changes its list from time to time), so your project will instead use the somewhat larger dictionary in the provided file **EnglishWords.txt**.

To get a sense of how the puzzle works, you should try to find the legal words in the puzzle shown on this page before looking at the solution in Figure 1.

Figure 1. The solution to the Spelling Bee puzzle from September 4, 2019

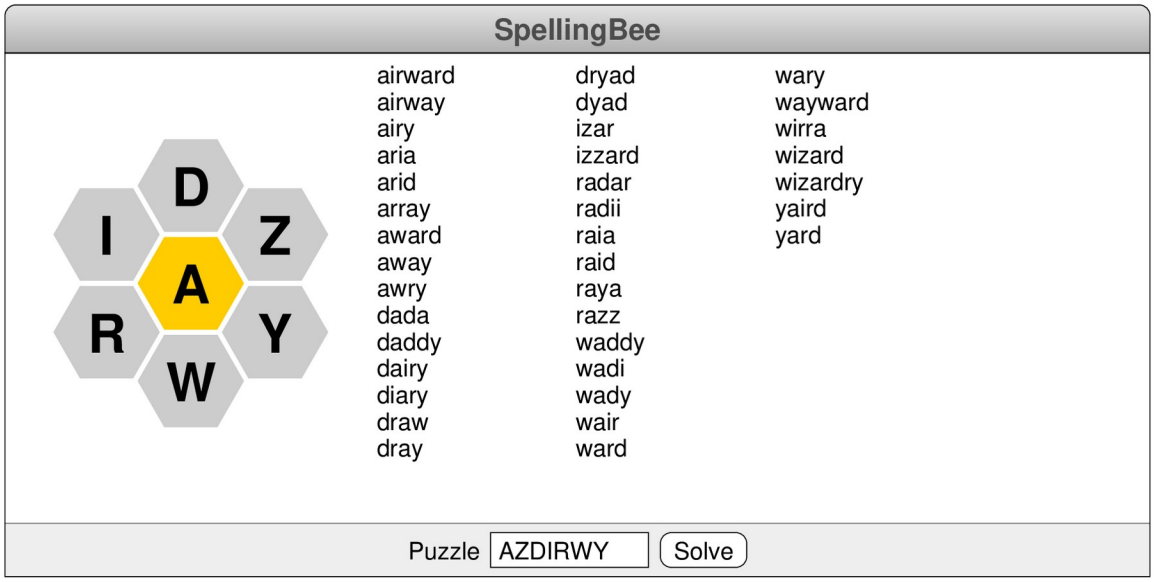


Figure 1 shows a screen shot of the solution to the SpellingBee puzzle using the letters shown on the previous page. In addition to the beehive-shaped diagram, the screen shows several columns listing the words one can find in this puzzle. The bottom of the window displays the controls available to the user. To begin a puzzle the user types a sequence of seven letters into the **Puzzle** text field and then hits the RETURN/ENTER key. Then those letters appear in the beehive hexagons, with the first letter in the center. There are rules for these letters also (later in the document). If the user clicks on the **Solve** button, the application - ***once you have implemented this part*** - will list the legal words that appear in that beehive puzzle.

Every puzzle on the *New York Times* website always includes at least one word that uses all seven letters in the puzzle. These words are called ***pangrams*** and score extra points. The pangram in the puzzle shown in Figure 1 is the word **wizardry**.

Because the focus of this assignment is on the string and file processing rather than on creating the graphical display, the parts of the assignment that draw the beehive and list the words on the screen are provided for you as a library class called ***SpellingBeeGraphics***. You are not required to learn or mess with this code unless you choose to. The Java 2D Graphics functionality has been abstracted to a point that is very easy to use, without diving into the details. **Feel free to explore and ask questions though**

In order to interact with the ***SpellingBeeGraphics*** class, you will first create a ***SpellingBeeGraphics*** instance (object) using the following line:

SpellingBeeGraphics sbg = SpellingBeeGraphics(); // create instance of class

The "reference variable" sbg now holds a reference (memory address) to a SpellingBeeGraphics object that takes care of all the graphics for you. You interact with this object by invoking its methods, which are listed in Figure 2 on the next page. The program included in the starter folder, for example, adds the **Puzzle** field and the **Solve** button to the bottom of the window. It even provides a minimal implementation of the **Puzzle** field so that you can set the puzzle letters.

The **Solve** button, however, requires some work on your part.

Methods in the SpellingBeeGraphics class. You will call these methods via a SpellingBeeGraphics object (like the **sbg** object created above)

Example: String value = sbg.getField("puzzle"); // get the text from the PUZZLE text field

addButton(*name*, *listener*) Adds a *pressable button* to the control area of the window with the label *name*. When the user clicks the button, the application invokes the *listener* function, passing the name of the button as a parameter. You will associate actions with the button press
addField(*name*, *listener*, *nchars*) Adds a text field to the control area of the window labeled with the string *name*. When the user enters a string into the field and hits the RETURN/ENTER key, the application invokes the *listener* function, **passing the contents of the field as a parameter**. The optional *nchars* (number of characters) parameter sets the width of the text field so that it can hold that many characters.
getField(*name*) Returns the value entered in the text field with the specified name.
setField(*name*, *value*) Sets the value of the named text field to the specified value.
getBeehiveLetters() Returns a string containing the seven letters in the beehive.
setBeehiveLetters(*letters*) Sets the letters in the beehive to the characters in the string letters.
clearWordList() Removes all the words from the word list at the right side of the window.
addWord(*word*, *color*) Adds a word to the word list display. The optional *color* parameter allows the caller to set the color. For example, you can supply Color.Blue here to display a pangram.
showMessage(*msg*, *color*) Displays the string *msg* in the message area at the bottom of the window. The optional *color* parameter allows the caller to set the color. You can clear the message area by calling showMessage("").
clearField(*name)* Clears the text contained in the specified field. And sets focus

SpellingBeeGraphics API Specification: In the starter code folder you will find a sub-folder called ***API_Specification.*** This folder contains a local web page app that contains API specs similar to the Java API site. Use this documentation by double clicking the ***index.html*** file. It will load in your browser and you can use it just like the Java site. This documentation was created with the ***javadoc*** utility and a special commenting style that you can see in the code. This is also how VS Code parses intellisense hover hints in the editor. Read about it here: https://www.baeldung.com/javadoc

This is only to expose you to the concepts. At this time, you are not expected to create this, but it will be coming soon.

The starter version of the **SpellingBee.java** file appears in Figure 3 on the next page. The main program consists of the following statements:

```
sbg = new SpellingBeeGraphics();                // creates an instance of the class
sbg.addField("Puzzle",  s -> puzzleAction(s));  // calls a method to add a field to the display
sbg.addButton("Solve", s -> solveAction());     // calls a method to add a button to the display
```

The first line creates the SpellingBeeGraphics object, which is then stored in the reference variable **sbg** so that it can be used in the rest of the class definition.

The next two lines add two *interactors* to the control strip at the bottom of the window: a text field labeled "Puzzle" and a button labeled "Solve". The calls to **addField** and **addButton** also specify the actions that occur when the user hits the RETURN key in the field or clicks on the button. The high-level explanation of what occurs is that hitting RETURN in the **Puzzle** field triggers a call to the ***puzzleAction*** method, passing in the puzzle string, and that clicking the **Solve** button triggers a call to the ***solveAction*** method. As it is defined, the ***solveAction*** method does not receive any arguments. It can get the puzzle letters by using some of the methods in the API.

Understanding the details of what is going on and interpreting the Java syntax used to specify the response require a bit more explanation and you do not need these details to successfully finish this lab. I am more than willing to discuss them though and I hope you'll take advantage of this research/learning opportunity.

All of your work will go in the file: **SpellingBee.java**

The interactivity required for this application is implemented using ***callback functions***, which are functions supplied by a client to a library that the library can later call to execute an operation on the client's behalf.

For example, the run method makes the following call to create the **Solve** button and register that the SpellingBee application should be notified whenever the user clicks that button:

```
sbg.addButton("Solve", (s) -> solveAction());
```

This ensures that when the button named **Solve** is pressed, the method named **solveAction** will be called. 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 the **addButton** method to determine the type of function it expects. In this case, that definition tells the compiler that **addButton** requires a function that takes a string

and returns no value. The argument **(s) → solveAction()** matches that definition and produces a function that takes a string as its argument and then calls the **solveAction** method, ignoring the value of the string s, which is not needed by **solveAction**. Notice in this situation that the solveAction method doesn't actually use the argument. Other methods that you define will use the argument. You can see this with the **puzzleAction** method, which requires the string of puzzle characters.

Task #1: Initialize the beehive with the letters in the puzzle field

Whenever you are faced with a large programming project, the most effective strategy is to define a series of tasks 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, even if the code you write to test a particular milestone doesn't make its way into the finished project. The advantage you get from making it possible to test each stage more than compensates for having to write a little extra code along the way.

The first task requires you to update the **puzzleAction** method so that typing seven letters into the **Puzzle** field and hitting RETURN updates the letters in the beehive on the screen, after first checking to see that the letters represent a legal puzzle, which must meet the following conditions:

- The puzzle must contain exactly seven characters.
- Every character must be one of the 26 letters.
- No letter may appear more than once in the puzzle.

If the string of characters entered by the user meets these criteria, your implementation of **puzzleAction** should call the **setBeehiveLetters** method so that the letters appear on the screen. If not, your code should call **showMessage** with a message telling the user why the puzzle is not properly formed. Prove to yourself that this works before moving on to implement the puzzle rules.

As you develop the code for the puzzle, you should be on the lookout for ways to decompose the problem into small, easily understood pieces. As an example, defining a method to check whether a puzzle is legal would almost certainly make your code for Task #1 easier to understand.

Task #2: Display the legal words in the SpellingBee puzzle

The second milestone represents most of the work necessary for solving the Spelling Bee puzzle. All you have to do is re-implement the **solveAction** method so that it goes through the dictionary and checks each word to see whether it appears in the puzzle if you follow all the legal rules from page 1.

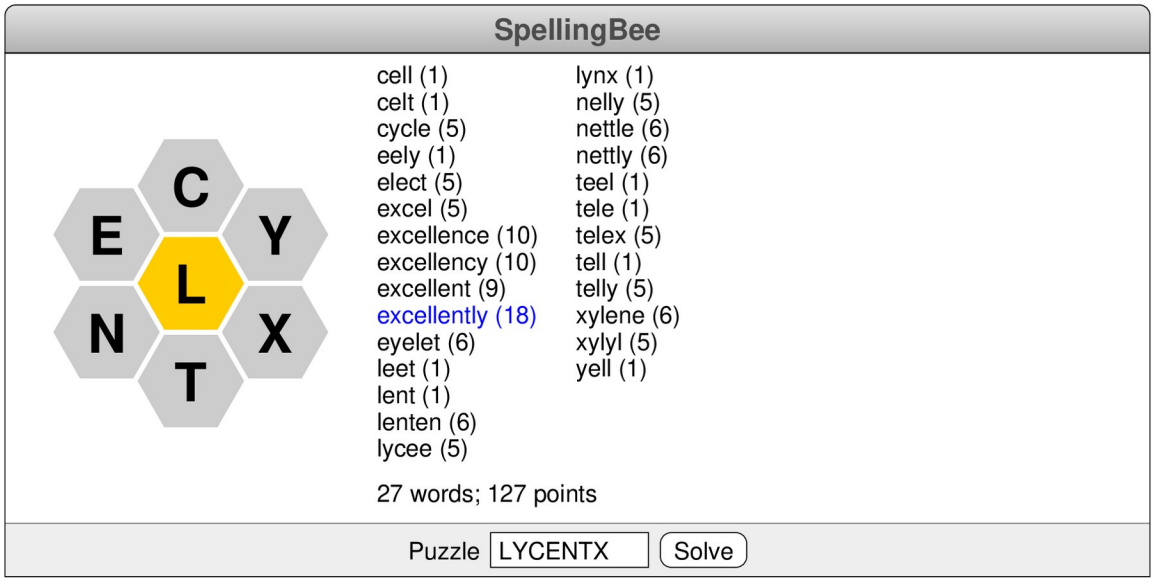
Since you are iterating through dictionary words, the only conditions you have left to check are the following:

- The word is at least four letters long.
- The word does not contain any letters other than the seven letters in the puzzle.
- The word contains the center letter, which appears at the start of the puzzle string.

Each time you find a word that fits these rules, you need to call the **addWord** method to ensure that it appears on the screen. Even though the code for this milestone is short, decomposition still makes sense. One possible decomposition involves writing the following functions:

- A function to read the dictionary from the **EnglishWords.txt** data file into a list. Use Java's ArrayList for this task. You can create one to store strings in the following way: **ArrayList<String> words = new ArrayList<String>();**
- A function to check whether a particular word meets the requirements
- A higher-level function that reads the dictionary and then checks each word

Figure 4. The output of Task #3, which includes scores



Task #3: Add scores to the display

For this milestone, your job is to extend the implementation of **solveAction** so that the scores displayed on the screen are followed by the score for that word in parentheses. In the online version of SpellingBee, a four-letter word is worth one point, but longer words score the number of letters they contain, so that a five-letter word is worth five points, a six-letter word is worth six points, and so on. Pangrams that use all seven letters in the puzzle score a bonus of seven points. Your code should also keep track of the number of words found and the total score and then displayed using the **showMessage** method. For example, the output for the puzzle "LYCENTX" (which appeared on January 8, 2020) should look appear as shown in Figure 4. The word excellently, for example, scores 18 points: 11 for the number of letters in the word and 7 for the pangram bonus. Note that the pangram appears in blue.

Task #4: Let the user try to find the words

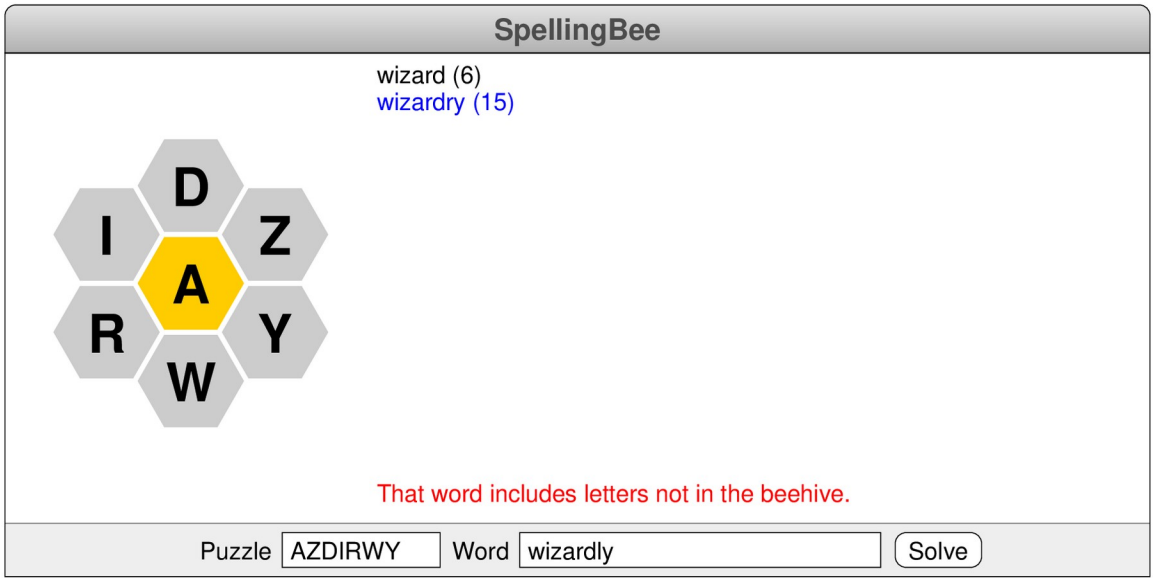
The online version of SpellingBee lets the user try to find the words rather than having the computer solve the puzzle. For your final milestone, add a new field labeled "Word" to the control strip and then implement a callback function that checks to see whether that word appears in the puzzle. If so, the SpellingBee application should add it to the word list, along with its score. If not, the application should use the **showMessage** method to tell the user what is wrong with the word.

The reasons for rejecting a word are:

- The word is not in the dictionary.
- The word includes letters not in the beehive.

- The word does not include at least four letters.
- The word does not include the center letter.
- The user has already found the word and is not allowed to score it twice.

Figure 5. Example of Task #4



When the user finds an acceptable word, the program should display the word and the score, just as it does when the computer solves the game. For example, Figure 5 shows the state of the program after the user has correctly found the words **wizard** and **wizardry** but then tried the word **wizardly**. It also improves the user experience if the program clears the **Word** field after every acceptable word so that the new word starts afresh. The **Solve** button should continue to work in Task #4 and should add in any words that the user missed.

The individual pieces of code you have to write for Task #4 are not particularly long or complex. Much of what makes this milestone challenging lies in integrating the new code with what you have written for the earlier milestones, particularly when you discover that you need to change the structure of your code. For example, Task #4 requires you to update the score whenever the user enters an acceptable word. For Task #3, you probably computed the score only at the end of the game. This change in the way the application works requires you to pull that part of the code out and put it in a separate method that you can call both after each user word and at the end.

Making this type of change during the development of a program is called ***refactoring,*** which is a critical activity in modern software engineering.

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 beehive diagram should all be uppercase, but the words in the English lexicon and

the word list displayed on the screen are all lowercase. At some point, your code will have to apply the necessary case conversions.

Task #5: Implement the shuffle button. The SpellingBee implementation on the *New York Times* site includes a button that shuffles the letters in the ***outer hexagons***. Doing so sometimes makes it easier to find the words.

Possible extensions

- *Generate the puzzle word.* In the SpellingBee solver you create for this assignment, the user is responsible for entering the seven-letter puzzle string. It would be fun to try and generate letter combinations that make a good puzzle. Puzzles must include at least one pangram but should probably not produce word lists that are too large. According to the website **https://nytbee.com**, the number of words in the *New York Times* puzzles has varied between 21 and 81, and the total number of points has ranged from 50 to 444. The *New York Times* also reduces the number of words by avoiding including the **S** character in the puzzles.
- *Implement a high score list.* Allow users to enter their names (or initials) and record their score and their place among the scores.
- *Find interesting SpellingBee puzzles.* You may want to think about how you might find SpellingBee puzzles that are interesting in some way. For example, the puzzle with the lowest total score (at least in the dictionary you're using) is ---------->

  which generates only one word, the somewhat archaic word **princox**, which appears in Shakespeare's *Romeo and Juliet.* It is a pangram, so it meets that requirement for a legal puzzle, but there aren't any other words. Similarly, you might want to find the puzzles that generate the longest pangrams. There are, for example, several puzzles that have 13-letter pangrams but no shorter ones. The challenging part of this idea is figuring out how to use the computer to find such puzzles.

Submission: Push your modified ***SpellingBee.java*** file

RUBRIC: 30 points total

Requirement	Points
Task 1 solved correctly	5
Task 2 solved correctly	5
Task 3 solved correctly	5
Task 4 solved correctly	5
Task 5 solved correctly	2
Appropriate Decomposition, no redundancy	5
Appropriate documentation	3