Eric Roberts and Jed Rembold Nifty Assignments, 2022

Spelling Bee Project

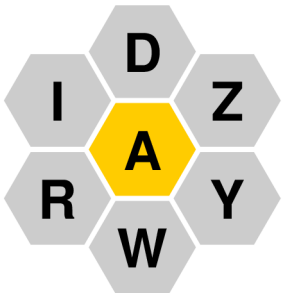
The purpose of this project is twofold:

1. To let you practice working with strings, array lists, and text files in Java.
2. To give you a chance to implement an exciting (and commercially successful) puzzle that emphasizes interactivity.

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:

- 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.
- Each word must not 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.
- Each word must contain the center letter at least once, which rules out the word `WIRY`.
- 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 file **EnglishWords.txt**, which is included in the starter project.

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

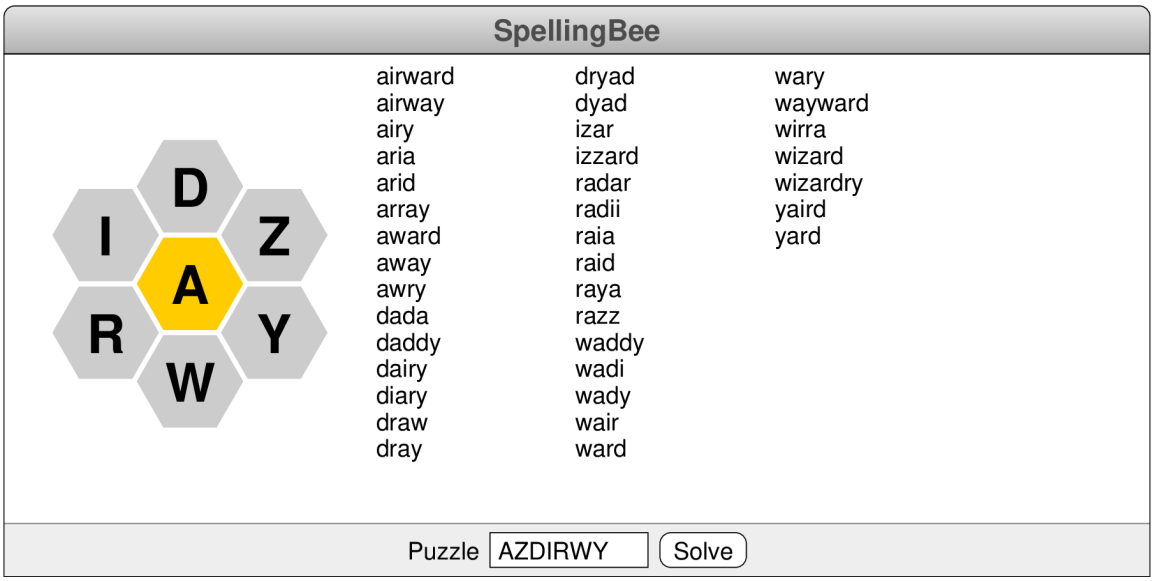


Figure 1 shows a screen image of the solution to the SpellingBee puzzle using the letters shown on the previous page. In addition to the beehive-shaped diagram, the screen shows several columns listing the words one can find in this puzzle. The bottom of the window displays the controls available to the user. If the user types a sequence of seven letters into the **Puzzle** field and then types the RETURN key, those letters appear in the beehive hexagons, with the first letter in the center. 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 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`, which you can import using the following line:

```
import edu.willamette.cs1.spellingbee.SpellingBeeGraphics;
```

Once you have done so, you can create a `SpellingBeeGraphics` object using the following line:

```
SpellingBeeGraphics sbg = SpellingBeeGraphics();
```

The variable `sbg` now holds a reference to the `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.

Figure 2. Methods in the SpellingBeeGraphics class

`addButton(`*name*`, `*listener*`)` Adds a button to the control string 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.
`addField(`*name*`, `*listener*`, `*nchars*`)` Adds a text field to the control string labeled with the string *name*. When the user enters a string and hits the RETURN key, the application invokes the *listener* function, passing the contents of the field as a parameter. The optional *nchars* 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("").`

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 = SpellingBeeGraphics();
sbg.addField("Puzzle", (s) -> puzzleAction(s));
sbg.addButton("Solve", (s) -> solveAction());
```

The first line creates the `SpellingBeeGraphics` object, which is then stored in the instance 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 field labeled `"Puzzle"` and a button labeled `"Solve"`. The calls to `addField` and the `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 field triggers a call to the `puzzleAction` method, passing in the puzzle string, and that clicking the **Solve** button triggers a call to `solveAction`. Understanding the details of what is going on and interpreting the Java syntax used to specify the response require a bit more explanation.

Figure 3. The Java starter file for the SpellingBee project

```
import edu.willamette.cs1.spellingbee.SpellingBeeGraphics;
import java.awt.Color;

public class SpellingBee {

    public void run() {
        sbg = new SpellingBeeGraphics();
        sbg.addField("Puzzle", (s) -> puzzleAction(s));
        sbg.addButton("Solve", (s) -> solveAction());
    }

    private void puzzleAction(String s) {
        sbg.showMessage("puzzleAction is not yet implemented", Color.RED);
    }

    private void solveAction() {
        sbg.showMessage("solveAction is not yet implemented", Color.RED);
    }

/* Constants */

    private static final String ENGLISH_DICTIONARY = "EnglishWords.txt";

/* Private instance variables */

    private SpellingBeeGraphics sbg;

/* Startup code */

    public static void main(String[] args) {
        new SpellingBee().run();
    }

}
```

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());
```

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`.

Milestone #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 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, 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 milestone 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.

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 Milestone #1 easier to understand.

Milestone #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 reimplement 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:

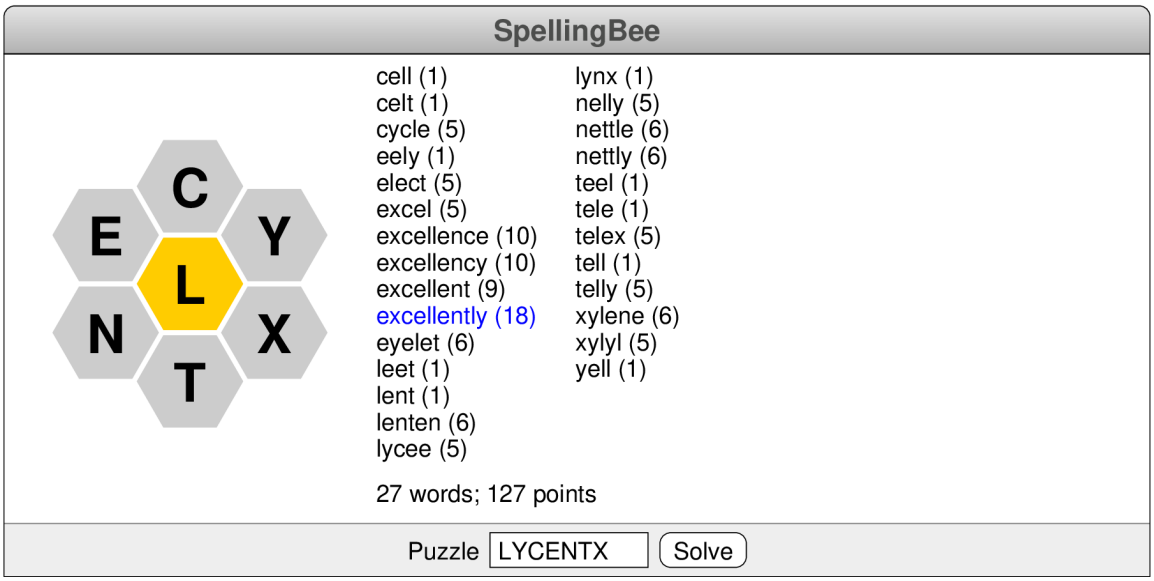
- The word is at least four letters long.
- The word does not contain any letters other than the seven letters in the puzzle.
- The word contains the center letter, which appears at the start of the puzzle string.

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
- 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 Milestone #3, which includes scores



Milestone #3: Add scores to the display

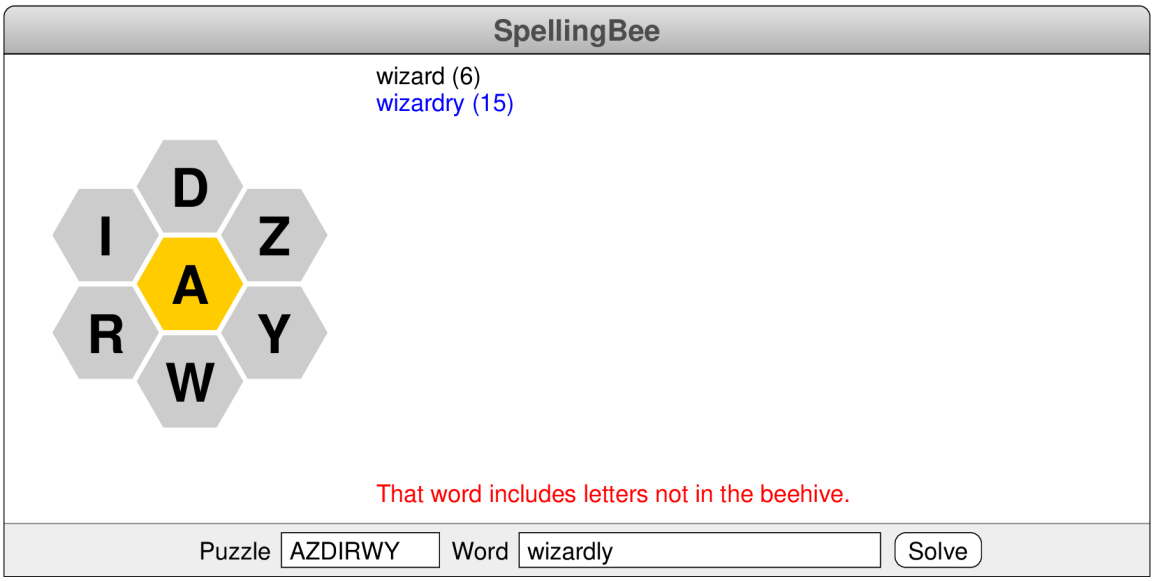
For this milestone, your job is to extend the implementation of `solveAction` so that the scores displayed on the screen are followed by the score for that word in parentheses. In the online version of SpellingBee, a four-letter word is worth one point, but longer words score the number of letters they 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.

Milestone #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 Milestone #4



When the user finds an acceptable word, the program should display the number of words 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 Milestone #4 and should add in any words that the user missed.

The individual pieces of code you have to write for Milestone #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, Milestone #4 requires you to update the score whenever the user enters an acceptable word. For Milestone #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 function 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.

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 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.
- *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.