Project 01: Betwordle

COSC 102 - Spring '25

Goal: In this project, you will implement a word game called *Betwordle* – a variation of the popular web game *Wordle*. In doing so, you'll better acclimate yourself to Java syntax and strengthen your programmatic design skills.

1 Overview

Betwordle's name is a portmanteau of *"between"* and *"Wordle"* – the latter being a web-based word game created by Josh Wardle where players guess a shared daily secret word. The game was purchased by the *New York Times* in 2022 and can be played for free online. *Betwordle* is a variant of Wordle featuring new rules and objectives.

1.1 Game Rules

The game plays as follows:

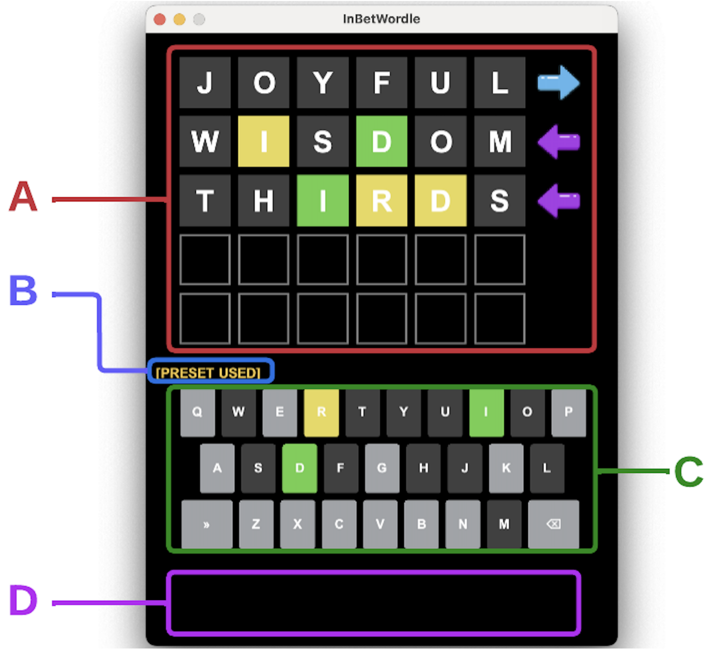
- Players take turns trying to guess a randomly chosen word.
- After a guess, players are given feedback regarding their guess's individual letters. Specifically, for each letter:
  - if the letter is in the secret word and in the correct spot, it is colored `green`
  - if the letter is in the secret word but not in the correct spot, it is colored `yellow`
  - if the letter is not in the secret word, it is colored **`dark gray`**
- The secret word and player's guesses must be actual, English words (*ex:* `"ZXVQR"` would not be a valid guess).

Betwordle has the following additional rules (differentiating it from standard Wordle):

- The randomly chosen secret word is **six characters long**, and the player has **five guesses** to identify it.
- After each guess, either a **left** or **right** arrow is displayed indicating if the secret word comes alphabetically **before** or **after** the guessed word, respectively.
- Related to above, each guess must be in the appropriate alphabetic range per the previous guess's arrows. *(For example, if a previous guess of **`DAWDLE`** displayed a right arrow, a guess of **`BRONZE`** would be rejected)*

1.2 The GUI

Betwordle makes use of a *Graphical User Interface* (GUI). This means the program runs in a separate window with a graphical abstraction, and allows the user to click buttons to enter their words (or use their physical keyboard).



All of the GUI code in this project is **already implemented for you**. On the next page is a breakdown of the GUI:

- **A:** the *game board* where the player's guesses and subsequent feedback are displayed. The board consists of **five rows** each containing **six cells** and (eventually) an **arrow**. Each cell will contain a **single letter**.
- **B:** the *debug toggle notifications* which display any activated *debug toggles*. These toggles are options which can be enabled/disabled to help you in your implementation (more on these below).
- **C:** the *graphical keyboard interface* which users can click to enter guesses. Keys on the keyboard change color to reflect the game board. The bottom left and right buttons are mapped to `enter` and `backspace` respectively.
- **D:** the *dialogue area* where messages are displayed to the user (*ex: "Game Over!"*).

1.3 Game Flow Breakdown

Using the GUI screenshot above, we can see an example of the game logic outlined in *Section 1.1*:

- In this instance the secret word is **`RAIDER`**.
- The player's first guess, **`JOYFUL`**, gets entered into the first row. Upon pressing *enter* the word is evaluated.

  None of the guess's letters, `J`, `O`, `Y`, `F`, `U`, or `L`, appear in `RAIDER`, thus the cells of the row and their respective keyboard keys are colored **`gray`**. Since `RAIDER` comes *after* `JOYFUL` alphabetically, a **right arrow** is displayed.
- For the next guess, **`WISDOM`**, the `D` appears in the same position as `RAIDER`, so its cell and keyboard key are colored **`green`**. The `I` appears in `RAIDER` but in a different position, so its cell and key are colored **`yellow`**. The row's remaining cells/keys are colored **`gray`**. A **left arrow** is displayed since `RAIDER` comes before `WISDOM`.
- for the third guess, **`THIRDS`**:
  - the `I` character is in the proper position, thus its cell is colored **`green`**. The `I` keyboard key was previously colored **`yellow`**, but now changes to be colored **`green`**.
  - The `D` character is in the wrong place, so its cell is colored **`yellow`**. However, since its keyboard key is colored **`green`** from a previous guess, its color **does not** change. The `R` cell and key are colored **`yellow`**.
  - the remaining `T`, `H`, and `S` cell and keyboard keys are turned **`gray`**.
  - `RAIDER` once again comes before `THIRDS` alphabetically, so a **left arrow** is displayed.

1.4 Text Files

The *Betwordle* program reads in **two different** text (`.txt`) files containing six-letter words when it is ran:

- A **secret words** file from which each game's secret word is randomly chosen from
- A **dictionary** file containing all the English six-letter words (which the player's guesses are validated against)

Furthermore, text files must adhere to the following formatting requirements:

For *both* the secret words and dictionary files:

- the **first line** of the file must contain only a number representing the **number of words** in the file
- each line must contain only a **single six-letter word**

For *only* the dictionary file:

- words must be in **ascending alphabetic order** (meaning `a` to `z`)

Example secret and dictionary words files are provided as **`secrets_words.txt`** and **`dictionary.txt`** respectively.

2 Provided Code

Provided to you are **three** `.java` files, though **you only need to add code to one of these**. Details on the next page:

2.1 *Betwordle* Launcher

The **`BetwordleLauncher.java`** file contains the `main` method used to launch the game.

At the top of this file are two `final boolean` variables used for **debug toggles**, which affect the game by changing randomization or displaying information to make your testing easier. When a toggle is enabled, a notification is displayed in the game window (area **B** in the diagram above).

Read the comments for each variable to understand the effect that each toggle has. You **do not need to modify** this file aside from changing the values of the debug toggle variables.

2.2 Game Logic

The **`GameLogic.java`** file contains all of the back-end logic for reacting to user input, evaluating guesses, and controlling the state of the game. **All of your code for this project will go in this file**.

This class contains **three functions** which you must modify (though you will **add more** helper functions):

- **`public static char[] initGame():`** short for "initialize". This function gets called **once** when the game is launched, before any other game code is ran. This function returns the secret word to be used as a `char` **array**.
- **`public static void reactToKeyPress(char key):`** this function is called automatically when the user presses a valid key (either on their physical keyboard or the graphical keyboard). The argument represents the key pressed. Invalid keys (such as numbers or symbols) are ignored by the game and this function is not called.
- **`public static void warmup():`** like `initGame()`, this function is called **once** when the game is launched. You will add code here as a warmup exercise to acclimate you to the code base (see *Section 3.1.1 Step 2*).

Additionally, there are numerous **final variables** declared at the top of `GameLogic` that are very important in your implementation. Review these variables and their comments to get an understanding of their utility.

Lastly, you are **not allowed** to add any more non-final global variables to `GameLogic.java`. However, you are welcome (and encouraged) to add additional `final` variables, but **only** if they are primitive types.

2.3 Game GUI

The **`GameGUI.java`** file contains all of the GUI code for the game, including drawing the game window and recognizing user input. **You cannot modify this file**.

That said, you *will* need to call several of the functions here. There are **twelve** functions you may need to call in this file (though you may not need all of them); they are listed below:

1. `public static char[] getSecretAsArr()`
2. `public static String getSecretAsStr()`
3. `public static char getBoardChar(int row, int col)`
4. `public static void setBoardChar(int row, int col, char letter)`
5. `public static Color getBoardColor(int row, int col)`
6. `public static void setBoardColor(int row, int col, Color newColor)`
7. `public static Color getKBColor(char key)`
8. `public static void setKBColor(char key, Color newColor)`
9. `public static void wiggleRow(int row)`
10. `public static int getArrowDirection(int row)`
11. `public static void setArrow(int row, int direction)`
12. `public static void triggerGameOver(boolean didPlayerWin)`

All twelve functions are located near the top of the file – read each function's comments to understand their utility.

3 Your Task

Your task is to implement the back-end game logic for the *Betwordle* game, including responding to player input, evaluating guesses by coloring cells/keys appropriately, and determining when a game over state is reached. Your implementation is broken up into **two milestones**; you will submit your code after each and receive feedback.

Remember: for both milestones you **may not add any additional non-final global variables** to any of the provided classes! That said, you are welcome and encouraged to add any **primitive-type** finals you like.

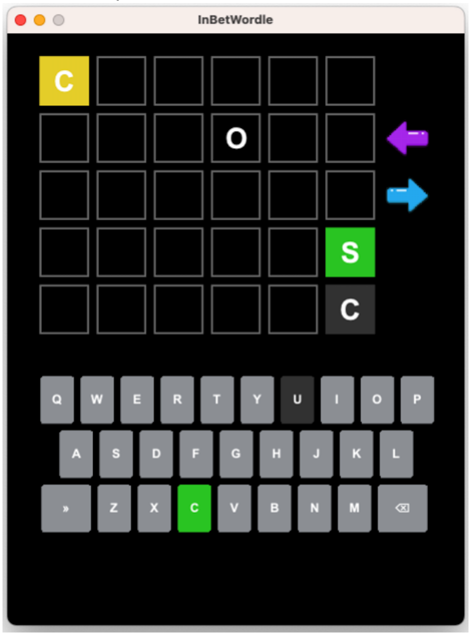
3.1 Milestone #1 (Warmup and Basic Functionality)

This first milestone will acclimate you to the code base and have you implement basic functionality. The goal is to make the game playable with **`RAIDER`** as the hard-coded secret word (though lacking some features/advanced logic).

3.1.1 Warmup

To get started, complete the following:

1. When you first attempt to launch the game, *Betwordle* will **crash with an exception** which you must fix. Read the exception, trace the code, and review the relevant comments to help identify this issue. Remember, the goal of the first milestone is to get the game working using the hard-coded secret word (default: `RAIDER`).
2. When the previous step is complete, the game should launch a window with an empty game grid that does not react to user input. **In the `warmup()` function** of `GameLogic` add code so that, when the game is first launched, the game's grid and keyboard look exactly like below:



3. Make it so that whenever the user presses "`P`", either via their physical keyboard or the graphical keyboard, the fourth row from the top (containing the green "`S`") "wiggles" (meaning it quickly shakes left and right).
   *Note: the code to complete this step will **not** go in the `warmup()` function.*
4. Once the previous tasks are complete, **comment out** the code you implemented for **steps 2 and 3** above. You **must include** the commented code in your Milestone #1 submission.

3.1.2 Basic Functionality

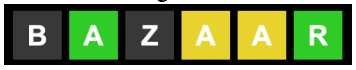
Next you will implement the basic game functionality, so the game will be playable from start to finish with some concessions. This functionality is outlined on the next page.

Your basic implementation must:

- use the hard-coded secret word (as defined by `HARDCODED_SECRET` – default: **`RAIDER`**)
- Properly react to player keyboard input to draw and delete characters on the game board. This includes reacting properly in all edge case scenarios (ex: typing more than six characters).
- Evaluate player guesses when *enter* is pressed by coloring the cells and keyboard keys appropriately. Keyboard colors should prioritize correctly (ex: a green key shouldn't change to yellow). While you don't need to validate guesses are real English words, there's one scenario where pressing *enter* should be rejected with a wiggle.
- End the game when the player guesses the word or runs out of guesses, displaying appropriate game over text.
- Under no circumstances should any player inputs result in an unhandled exception.

Your basic implementation does **not** yet need to:

- pick a random secret word from a file
- validate that player guesses are actual English words
- display left or right arrows after guesses nor prevent invalid guesses based on previous arrows/alphabet ranges
- properly handle duplicate letters; it can simply color letters depending on if they're in the word and/or in the correct place. Given the secret word of **`RAIDER`**, a guess of **`BAZAAR`** should color like so:



Your Milestone #1 code submission must include only the above functionality.
Do not include Milestone #2 functionality in your Milestone #1 code submission!

3.2 Milestone #2 (Advanced Features and Logic)

Your final code must implement the advanced features described below.

3.2.1 Reading Dictionary Files

Your *Betwordle* program must read in two text files: a **secret words** and **dictionary** file. A random secret word is selected from the *secret words* file, and the *dictionary* file should be used to validate guesses are real, English words.

Your code should **not read through either file more than once**. If either file is not found or violates any of the formatting requirements (see *Section 1.4*), your game must print an error message then terminate using **`System.exit(1);`**.

Note: your game must work with any properly formatted secret word and/or dictionary file – not just the ones provided.

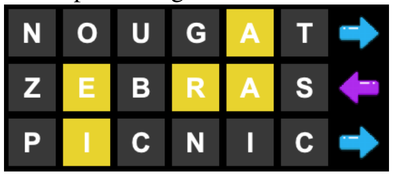
3.2.2 Guess Arrow Feedback

After each guess entered, either a **left** or **right** arrow is displayed indicating if the secret word comes alphabetically **before** or **after** the player's guess, respectively. No arrow is displayed when the secret word is correctly guessed.

3.2.3 Advanced Guess Validation

A player guess should be rejected in either of the two additional scenarios below:

- if the guess is not an actual, English word (per the supplied dictionary text file)
- if the guess is not in the appropriate alphabetic range per the previous guesses' arrows.
  For example, given a scenario where the previous guesses are as follows:

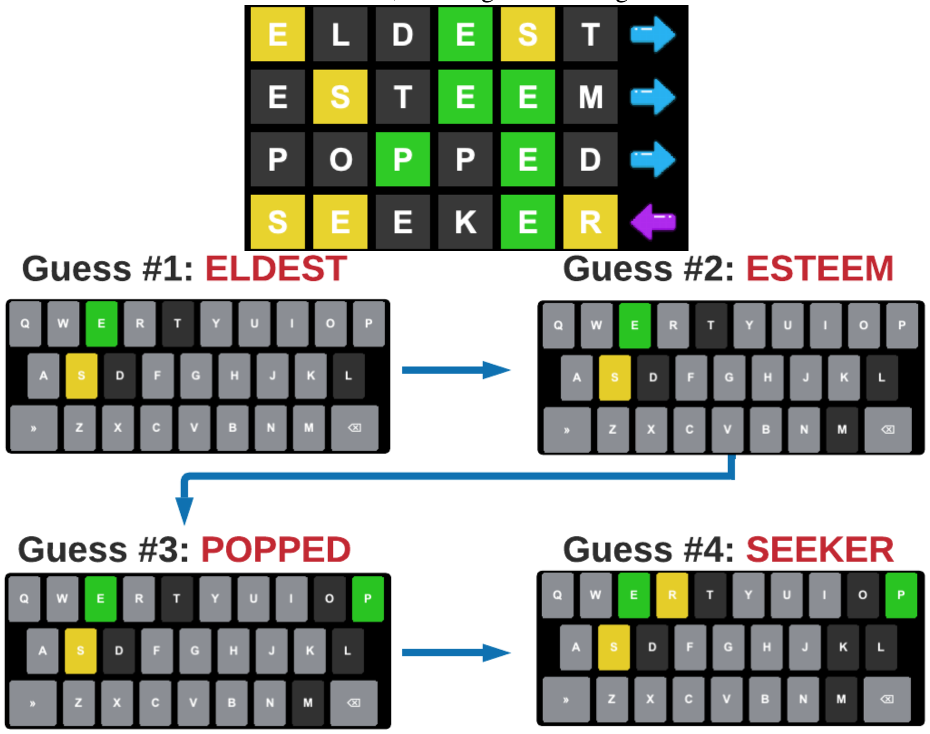


  Any guess that is alphabetically *before* **`PICNIC`** or *after* **`ZEBRAS`** is rejected. Though **`NOUGAT`** was entered first, the start of the valid alphabetic range is defined by later guess **`PICNIC`** (thus a guess of **`ORCHID`** would be invalid).

An invalid guess should trigger a wiggle, but the entered letters in the current row should **not** be cleared. After the first guess, you **should not** be searching through the entirety of the valid words to validate guesses (review your globals!).

3.2.4 Duplicate Letter Handling

The official Wordle game has particular rules when evaluating words/guesses with duplicate letters, which your *Betwordle* will match. Review the screenshot below, showing a series of guesses where the secret word is **`RUPEES`**:



From the above screenshots, you will be able to reverse-engineer how the duplicate letter cell/keyboard coloring works.

3.2.5 Debug Toggles

Lastly, both of the debug toggles in `BetwordleLauncher` should work properly. The functionality for one of these is already implemented for you and you must integrate the logic for the other in `GameLogic.java`.

4 Tips

- Though your `GameLogic.java` contains only three functions, it is expected that you will add **many more** helper functions in the design of your implementation. You will be graded on the quality of your design; having long, bloated, difficult to read and unintuitive functions will adversely affect your grade!

  Remember your **SOFA** principles (if we haven't covered this yet, we soon will!) – keep your functions short and singularly focused. Consider helper functions for small, generic tasks that you need to repeat multiple times in different contexts (ex: searching an array for a value). Think about levels of abstraction too!
- Make extensive use of your debug toggles; they will be very helpful in your testing and debugging processes.
- Lastly, though `Strings` can be useful here, remember that they have much greater memory/performance overhead than the primitive `char` type. If you can use a `char` instead of a `String`, you should!

5 Submission

You will submit **only your `GameLogic.java`** to the link on our course Moodle page after the completion of each Milestone – do not submit any other code or text files. The milestones are due as follows:

- **Milestone #1:** due **Thursday, February 20th** at **9:00PM**.
- **Milestone #2:** due **Thursday, March 6th** at **9:00PM**.