"RUBLE"

RUBY

+

WORDLE

TERMINAL APP

BY

CHRIS

HULLMAN

WHAT IS RUBLE?

Ruble is a Ruby developed terminal app replica of the real-world word game called "Wordle", of which can be played here: https://www.nytimes.com/games/wordle/index.html (viewed 18/04/2022)

How to play Ruble:

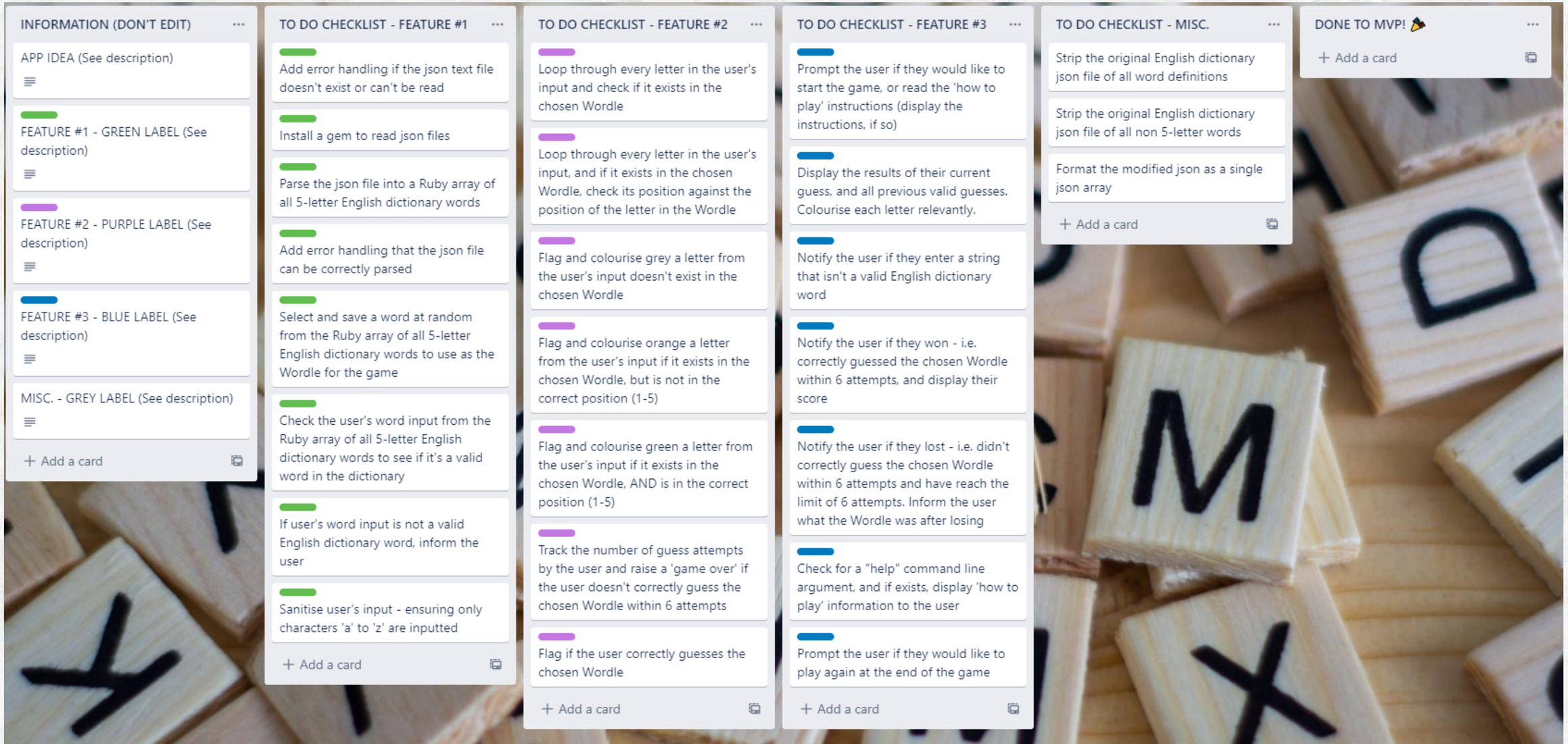
You will have **6** attempts to guess a randomly selected, valid **5 letter** English word, known as the **Wordle**, by inputting letters 'a' to 'z' to form a valid 5 letter English word as your guess. For every guess you make, you will get the following feedback. For example, you guess "hello":

H E L L O

- The letter H is both in the Wordle, and in the correct spot (so marked GREEN)
- The letter L is in the Wordle, but is not in the correct spot (so marked ORANGE)
- The letters E and O are not in the Wordle (so marked BLACK or GREY)

TRELLO - IMPLEMENTATION PLAN

https://trello.com/b/V7q9Pmvl/ruble-t1a3



INFORMATION (DON'T EDIT)

- APP IDEA (See description)
- FEATURE #1 - GREEN LABEL (See description)
- FEATURE #2 - PURPLE LABEL (See description)
- FEATURE #3 - BLUE LABEL (See description)
- MISC. - GREY LABEL (See description)

+ Add a card

TO DO CHECKLIST - FEATURE #1

- Add error handling if the json text file doesn't exist or can't be read
- Install a gem to read json files
- Parse the json file into a Ruby array of all 5-letter English dictionary words
- Add error handling that the json file can be correctly parsed
- Select and save a word at random from the Ruby array of all 5-letter English dictionary words to use as the Wordle for the game
- Check the user's word input from the Ruby array of all 5-letter English dictionary words to see if it's a valid word in the dictionary
- If user's word input is not a valid English dictionary word, inform the user
- Sanitise user's input - ensuring only characters 'a' to 'z' are inputted

+ Add a card

TO DO CHECKLIST - FEATURE #2

- Loop through every letter in the user's input and check if it exists in the chosen Wordle
- Loop through every letter in the user's input, and if it exists in the chosen Wordle, check its position against the position of the letter in the Wordle
- Flag and colourise grey a letter from the user's input doesn't exist in the chosen Wordle
- Flag and colourise orange a letter from the user's input if it exists in the chosen Wordle, but is not in the correct position (1-5)
- Flag and colourise green a letter from the user's input if it exists in the chosen Wordle, AND is in the correct position (1-5)
- Track the number of guess attempts by the user and raise a 'game over' if the user doesn't correctly guess the chosen Wordle within 6 attempts
- Flag if the user correctly guesses the chosen Wordle

+ Add a card

TO DO CHECKLIST - FEATURE #3

- Prompt the user if they would like to start the game, or read the 'how to play' instructions (display the instructions, if so)
- Display the results of their current guess, and all previous valid guesses. Colourise each letter relevantly.
- Notify the user if they enter a string that isn't a valid English dictionary word
- Notify the user if they won - i.e. correctly guessed the chosen Wordle within 6 attempts, and display their score
- Notify the user if they lost - i.e. didn't correctly guess the chosen Wordle within 6 attempts and have reach the limit of 6 attempts. Inform the user what the Wordle was after losing
- Check for a "help" command line argument, and if exists, display 'how to play' information to the user
- Prompt the user if they would like to play again at the end of the game

+ Add a card

TO DO CHECKLIST - MISC.

- Strip the original English dictionary json file of all word definitions
- Strip the original English dictionary json file of all non 5-letter words
- Format the modified json as a single json array

+ Add a card

DONE TO MVP! 🎉

+ Add a card

MISC. FEATURE

About: The code for this feature needs to be actioned as a one-off for the game app, in order to strip an original source json or text file containing the entire English dictionary from all word definitions and all non 5 letter words, as that content isn't needed for the game app.

There are **three possible dictionary sources** that I had found, and only one will be used to extract the relevant 5 letter words:

- **From GitHub user matthewreagan** - https://raw.githubusercontent.com/matthewreagan/WebstersEnglishDictionary/master/dictionary_alpha_arrays.json

- **WordNet** - https://wordnet.princeton.edu/ (sourcing and modifying from WordNet 2.1 database files - FREE TO USE AND MODIFY - modification result here: https://raw.githubusercontent.com/chullman/ruble/e79e3d6ae63632ee63436fb47aa5b8651aff6c1a/src/misc_feature/all_words_from_wordnet_2_1.txt)

- **WordsAPI** (from Rapid API) - https://rapidapi.com/dpventures/api/wordsapi/

MISC. FEATURE

Code Logic Steps:

1) Query the remote/online based source of ALL dictionary words and their definitions (if included). We don't want to keep a local storage of all known dictionary words and definitions as this will make the app package size far too large in memory.

- This involves directly querying the remote file source URL or API URI string (as the HTTP 'Request'). If I choose to source from the WordsAPI API, I need a valid account with Rapid API and pass in my unique "API Key" in the code.
- Add error handling if the HTTP 'Response' is anything but a '200 OK' response code.

2) For a successful response, return in memory the entire response body as a string, store it in a variable, any extract only those words that are 5 letters in length, and ONLY include letters from 'a' to 'z'

3) Format the stripped string into valid JSON syntax (which is fortunately identical to Ruby array and hash syntax. Parse the newly formatted string as JSON using the JSON gem, so its words can then be sorted into alphabetical order and any duplicate words removed

3) Write the final JSON to a new local .json/.txt file for later use by the final app. Assumption in the code made that there is write access to the file directory of whoever is executing this Ruby code.

FEATURE #1

About: Read, parse, and process every 5-letter word contained in a text file containing every valid English word from a dictionary.

The external text (.json or .txt) file will be stored in a file directory somewhere with the Ruby files. Its content will be structured as a JSON format, for easier reading and parsing by the Ruby app.

There will be two occurrences why the text file will need to be read and processed:

- Selecting at complete random one word from the text file to use as the Wordle for the game.

- Checking the user's word input against every word in the text file to check if the user's input is a valid English dictionary word.

FEATURE #1

Code Logic Steps:

1) Create an object of the custom class 'FileHandler' and TRY to open the external local .json or .txt file containing all the 5-letter words. Raise errors if either:
- The file referenced could not be found on the specified directory
- Permission is denied in opening the file - which can be fixed by running "chmod +r file_name" if the user has permission to do so

2) Read the entire contents of the file, of which should be in a JSON string format. Raise an error if the content is empty.

3) Parse the string using the JSON gem. Raise errors if either:
- The string isn't a valid JSON string
- The JSON string is empty of content - e.g. "[]" (if array) or "{}" (if hash)

4) Sanitizing the user's input - converting it to lowercase, stripping all leading and trailing whitespace, ensuring that the input only contains the characters 'a' to 'z', ensuring that the input is 5-letters long.

5) Checking the user's input against the sourced memory of all valid 5-letter English words to see if it's a valid word.

6) Call a method to randomly select one from from the sourced memory of all valid 5-letter English words to use as the 'Wordle'.

FEATURE #2

About: Check the user's inputted word (assuming it's a valid English dictionary word) against the chosen Wordle to see how close the user's guess attempt was.

Note, be mindful of both user's inputted words and the chosen Wordle if either or both of them contain duplicate letters - e.g. like with the word "hello"

FEATURE #2

Code Logic Steps:

1) Create a class and an object called 'AnswerProcessor' and pass into it two strings - the user's inputted guess, and the chosen Wordle. With this object, have two loops or iterators - one to loop through every character in the user's inputted guess string, and one to loop through every character in the Wordle string. Then:

1a) Check for GREENS and flag it (possibly by storing the results in a hash). Then,

1b) Start the loops from the first letters, and check for ORANGES and flag them. Then,

2c) Start the loops from the first letters, and check for BLACKS/GREYS and flag them

2) Restrict the user to a maximum of 6 guess attempts: (use a counter variable to keep track of attempts)

- If all letters are flagged GREEN at or before 6 attempts - the game is won
- Otherwise, if 6 attempts are reached. The game is lost - inform the user of what the Wordle was.

FEATURE #3

About: Display accurate and informative output for the user's every guess attempt, including when they win or have lost. Add logic to handle the user starting the game, quitting the game, and displaying game instructions and/or help, and display informative information where relevant so as to not confuse the user.

Code Logic Steps:

1) Prompt the user if they would like to start the game or read the instructions/'how to play', and handle the selection accordingly.

2) Display the results of their current guess and all previous valid guesses. Colourise each letter relevantly (using 'Colorize' gem).

3) Notify the user if they enter an invalid word as their guess

4) Notify the user if they have won or if they reached the max of 6 guess attempts (either way, end the game accordingly, and provide the user the option to try again). If the user lost the game, tell the user what the correct Wordle was.

5) Check for a "help", "--help" or "-h" command line argument, and if so, handle it and display the instructions/'how to play' information to the user.

CODE CONSIDERATIONS

(Apart from applying good/accepted Ruby conventions in code naming, styling and formatting)

'Coupling' and global variables - meaning that I've tried my best to ensure that methods and classes are not programmatically dependent on one-another. Note with my following code, the methods 'try_file_open' and 'try_file_read' can be called independently even if the other method is removed. The only restriction is the argument that is required by the methods:

```ruby
class FileHandler

  def initialize
    @file_path = ""
  end

  def try_file_open(file_path)
    @file_path = file_path
    begin
      file = File.open(@file_path)

      return file

    rescue Errno::ENOENT => e

      puts "ERROR: No such file \"#{@file_path}\" found"
      puts "ERROR @: #{e.backtrace[1]}"
      try_file_close(file)
      return nil

    rescue Errno::EACCES => e

      puts "ERROR: Permission denied in opening the file #{@file_path}"
      puts "Ensure that everyone has read access to the file by running - chmod 444 #{@file_path}"
      puts "ERROR @: #{e.backtrace[1]}"
      try_file_close(file)
      return nil

    end
  end

  def try_file_read(file)

    file_contents = file.read

    try_file_close(file)

    file_contents.chomp!

    if file_contents.empty?
      raise ArgumentError, "ERROR: The file \"#{@file_path}\" is empty"
    end

    return file_contents

  end
```

CODE CONSIDERATIONS

Error handling - For example with my following code. As far as I'm aware, only the following two errors can arise when attempting to open a file in my code. That is, "ENOENT", meaning a file doesn't appear to exist, and "EACCES", meaning that the file was found, but unable to be opened due to permission restrictions. Note that regardless of the error, I try to close the file with my 'try_file_close' method in case somehow it was opened.

```ruby
def try_file_open(file_path)
  @file_path = file_path
  begin
    file = File.open(@file_path)

    return file

  rescue Errno::ENOENT => e

    puts "ERROR: No such file \"#{@file_path}\" found"
    puts "ERROR @: #{e.backtrace[1]}"
    try_file_close(file)
    return nil

  rescue Errno::EACCES => e

    puts "ERROR: Permission denied in opening the file #{@file_path}"
    puts "Ensure that everyone has read access to the file by running - chmod 444 #{@file_path}"
    puts "ERROR @: #{e.backtrace[1]}"
    try_file_close(file)
    return nil

  end
end
```

CODE CONSIDERATIONS

Abstractions - I've extended the String class, through the use of "monkey patching", as seen in the below code snippet, to abstract away the calling of the Colorize gem's 'colorize' method, but wrapping it a custom class called 'color'. This means in the perhaps very unlikely scenario that the methods for the Colorize gem change in the future, only one-line in my code needs to be changed. I'm not directly referencing the Colorize gem anywhere else in my code.

```ruby
require 'colorized_string'

module ColorizeStringPatches
    refine String do
        def color(text_color=:default, background_color=:default)
            return ColorizedString[self.to_s].colorize(:color => text_color, :background => background_color)
        end
    end
end
```

CODE CONSIDERATIONS

Maximum size of arrays in memory - In my Misc Feature of stripping an entire database source of English words and definitions into only 5-letter words, meant parsing the entire source into an array stored in memory to do this check. Luckily, Ruby seemed to handle it well, as least on my dev computer.

```ruby
def strip_to_custom_json(body)
    json_formatted = "["

    (body).split("\n").each do |line|
        words_in_a_line = line.split(" ")

        # explanation of this regex and the gsub method: https://stackoverflow.com/a/6344630
        special = "?<>',?[]}{=-)(*&^%$#`~{}_\""
        special_char_regex = /[#{special.gsub(/./){|char| "\\#{char}"}}]/

        words_in_a_line[0] = words_in_a_line[0].to_s

        if words_in_a_line[0].length == 5 && !(words_in_a_line[0].include?(" ")) && !(/\d/.match(words_in_a_line[0])) && !(words_in_a_line[0] =~ special_char_regex)
            json_formatted += "\"#{words_in_a_line[0]}\","
        end
    end

    # remove the very last comma char
    json_formatted.chop!

    json_formatted += "]"
    json_formatted = JSON.parse(json_formatted)
    json_formatted.uniq!
    json_formatted.sort!
    json_formatted
end
```

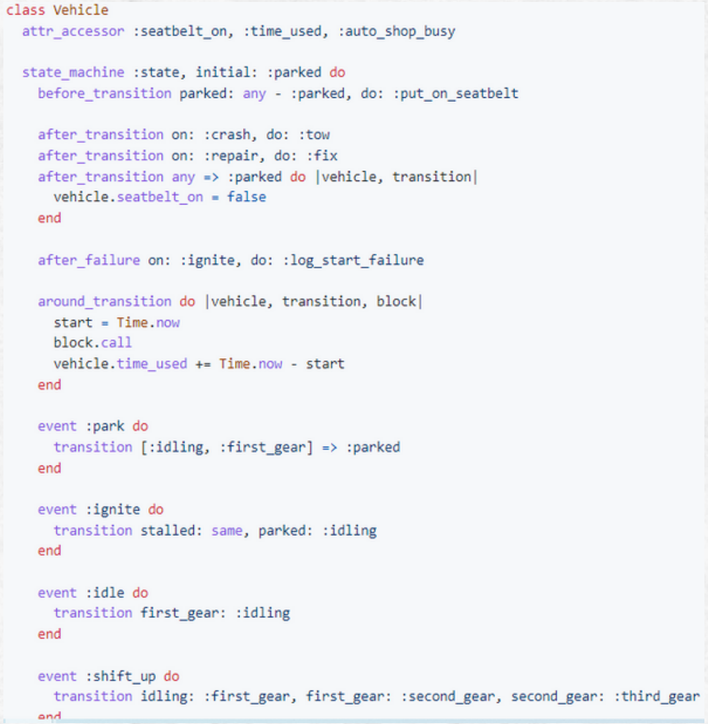
CODE CONSIDERATIONS

Preserving order of values in a hash - Apparently, prior to Ruby version 1.9.2, unlike with arrays, the order of which you add a key/value pair into a hash variable, may not necessarily be the same order that gets returned upon calling the hash (i.e. no deterministic order). So when I develop code to check the user's inputted word against the chosen Wordle, I have to store the word's letters in a hash like the following - so that it keeps a record of the letters' position. Note that each letter position is the key in the hash.

```
{1 => ["h", :green], 2 => ["e", :grey], 3 => ["l", :orange], 4 => ["l", :orange], 5 => ["o", :grey]}
```

CODE CONSIDERATIONS

Implementing a 'State Machine' - OPTIONAL - I MAY OR MAY NOT INCLUDE THIS IN MY FINAL GAME APP. I found a gem called 'state machines' that I think would be useful for my app. Source: https://github.com/state-machines/state_machines This would be handy to implement for my game logic to tie into Feature #3 to control and ensure correct order of gameplay. For example, by implementing state machines in my main execution file in Ruby, I could write code to ensure that game logic is handled in the correct order. For example, the restricting the code from going straight from the game's title menu selection directly to the 'game over' code logic, by means of "events" and "transitions". Like with the following sample code sourced from the gem's documentation:



```ruby
class Vehicle
  attr_accessor :seatbelt_on, :time_used, :auto_shop_busy

  state_machine :state, initial: :parked do
    before_transition parked: any - :parked, do: :put_on_seatbelt

    after_transition on: :crash, do: :tow
    after_transition on: :repair, do: :fix
    after_transition any => :parked do |vehicle, transition|
      vehicle.seatbelt_on = false
    end

    after_failure on: :ignite, do: :log_start_failure

    around_transition do |vehicle, transition, block|
      start = Time.now
      block.call
      vehicle.time_used += Time.now - start
    end

    event :park do
      transition [:idling, :first_gear] => :parked
    end

    event :ignite do
      transition stalled: same, parked: :idling
    end

    event :idle do
      transition first_gear: :idling
    end

    event :shift_up do
      transition idling: :first_gear, first_gear: :second_gear, second_gear: :third_gear
    end
```

CHALLENGES AND ETHICAL ISSUES

Challenges:

- Handling Wordles and/or input where there are duplicate letters (e.g. "hello" - two l's)

- Finding a free source dictionary that:
    - Doesn't include proper nouns - like names
    - Includes plurals like "girls" and "women". The real-world Wordle game accepts these as valid input for guesses, but my sourced dictionary doesn't

(Apparently, the real-world Wordle game sources from a hand-picked collection of words that are checked against for valid user guesses, as well as a seperate hand-picked collection of words that are selected to use as the daily Wordle. I have no time or scope to hand-pick 5-letter words for my Ruble game app to source.)

- Possible future feature: Allow the user to lookup the definition of the specific Wordle at the end of each game. But this would require the app to be internet-connected so such a lookup could occur.

Ethical Issues:

- Inputting vulgar/rude words and having it accepted by the app, or a rude word was chosen as the Wordle. Including euphemisms.
- Options for colour-blind users. My game app currently is vital that the user can see the colours of their guesses.
- Future feature: Add an option for user's to login to the app and store their score history, but this presents the possible ethical issue of the user storing their personal information like their real name, or a password that my app doesn't securely store.

FAVOURITE PARTS

- Sourcing a suitable dictionary of English words and trying to process it in different, optimised ways so it is suitable for my game app.