**Ghost**

Today we'll be writing a Ruby implementation of everyone's (or maybe just my) favorite road-trip word game, [Ghost](https://en.wikipedia.org/wiki/Ghost_(game)).

**Learning Goals**

* Understand how different classes interact
* Be able to write classes in different files and use require\_relative to connect them
* Know how to test methods in pry
* Know how to read lines from a text file
* Understand how \_\_FILE\_\_ == $PROGRAM\_NAME works

**Phase 1: Playing a Single Round**

Let's start by writing the logic to play a single round of Ghost (that is, playing until one player spells a word). Write your game for two players only, and don't worry about keeping track of wins/losses (we can get to this later). The basic logic will look something like this:

* Instantiate a new Game object, passing in both of the Players.
  + The Game maintains a fragment instance variable, which represents the word as it has been built up by the players.
  + The Players take turns adding a letter to the fragment. The Game should ensure that a play is valid before actually changing the fragment.
  + The Game checks the fragment against a dictionary; if the fragment is a word contained in the dictionary, then the previous player loses.

**Game**

#initialize

Assign instance variables for the players, fragment, and dictionary. Since we'll be checking the fragment for inclusion in the dictionary, we'll want to use a data structure with fast lookup: a Hash or [Set](http://ruby-doc.org/stdlib-2.4.2/libdoc/set/rdoc/Set.html) would be ideal. You can use [this file](https://assets.aaonline.io/fullstack/ruby/projects/ghost/dictionary.txt) to populate your dictionary; it contains only words three letters or longer (otherwise we wouldn't have a very interesting game).

💡 *NOTE:* Using a Hash or a Set instead of an Array to store our dictionary allows us to very quickly check if the fragment is included in the dictionary. In fact, the amount of time it would take to see if the fragment was included would be independent of how long the dictionary is because these data structures don't require you to examine every element when you are checking for inclusion. If we stored the dictionary just as an array of strings, using the Array#include? method would take longer as our array of strings got longer. Keep in mind that when you are then checking to see if there are any words in the dictionary that can be created by adding another letter to the fragment in your valid\_play? method, you are potentially looking at every word in the dictionary. Therefore, this operation will not be speed-boosted by our use of a Set or Hash.

#play\_round

The core game logic lives here. I wrote a number of helper methods to keep things clean:

* #current\_player
* #previous\_player
* #next\_player!: updates the current\_player and previous\_player
* #take\_turn(player): gets a string from the player until a valid play is made; then updates the fragment and checks against the dictionary. You may also want to alert the player if they attempt to make an invalid move (or, if you're feeling mean, you might cause them to lose outright).
* #valid\_play?(string): Checks that string is a letter of the alphabet and that there are words we can spell after adding it to the fragment

**Player**

I wrote initialize, guess, and alert\_invalid\_guess methods. You'll probably want each Player to have a name, as well.

**Phase 2: Playing a Full Game**

Now that we have the logic to play a single round of Ghost, we'll have to add another layer.

Game#losses**and**Game#record

In a game of Ghost, a player "earns" a letter each time they lose a round. Thus, if Eric beats Ryan 3 times and loses once, then Eric has a "G" and Ryan has a "GHO". If a player spells the word "GHOST", they are eliminated from play (and in the case of two players, the other player wins).

I added a losses hash to my Game class. The keys to the hash are Players, and the values are the number of games that player has lost. Update this at the end of #play\_round. For flavor, I also wrote a helper method, #record(player), that translates a player's losses into a substring of "GHOST".

Game#run

This method should call #play\_round until one of the players reaches 5 losses ("GHOST"). I wrote a helper method, #display\_standings, to show the scoreboard at the beginning of each round. Remember to reset the fragment at the beginning of each round, as well!

**Phase 3: Multiplayer!**

Refactor your game to work with more than just two players. Instead of ending the game when one of the players reaches five losses, simply exclude that player from further rounds. End the game when only one player is left standing. **Hint**: You won't be able to use an instance variable for each player anymore. What data structure might we use as an alternative?

**Phase Bonus**

* Write an AiPlayer class for your Ghost game. You'll need to figure out the logic for picking a winning letter on each turn. In order to do this, your AiPlayer will need to know both the current fragment and the number of other players (n).
  + If adding a letter to the fragment would spell a word, then the letter is a losing move.
  + If adding a letter to the fragment would leave only words with n or fewer additional letters as possibilities, then the letter is a winning move.
  + Your AI should take any available winning move; if none is available, randomly select a losing move.
    - See if you can improve your AI by computing the entire tree of possible moves from the current position. Choose the move that leaves the fewest losers and the most winners in the tree.