Due Sunday, February 10th at 11:59pm
Note: Question 7 removed.
Note: Edit made to final file naming.

HW 1: Ruby calisthenics

In this homework, you will do some simple programming exercises to get familiar with the Ruby language. We will provide detailed automatic (hopefully) and personalized grading of your code.

Skeleton code is provided in the attachments here for each part! Start with the skeleton code and read the comments carefully.

Half of the information needed for this assignment has not yet been covered in class but will be on Thursday. As of Tuesday's class, you should be able to complete #1-3. I encourage you to read ahead and get started on the material for Thursday.

NOTE: For all questions involving words or strings, you may assume that the definition of a "word" is "a sequence of characters whose boundaries are matched by the \b construct in Ruby regexps."

Submission: Submit each part as a separate .rb file in Piazza. Name each file based on this: **youruccsID_part_partnumber_answer.rb** Send me a **private** post including your 7 files. Select folder hw1 with summary "Homework 1".

Part 1: fun with strings

(a) Write a method that determines whether a given word or phrase is a palindrome, that is, it reads the same backwards as forwards, ignoring case, punctuation, and nonword characters. (a "nonword character" is defined for our purposes as "a character that Ruby regexps would treat as a nonword character".) Your solution shouldn't use loops or iteration of any kind. You will find regular-expression syntax very useful; it's reviewed briefly in the book, and the website rubular.com lets you try out Ruby regular expressions "live". Methods you might find useful (which you'll have to look up in Ruby documentation, ruby-doc.org) include: `String#downcase, String#gsub, String#reverse`
Suggestion: once you have your code working, consider making it more beautiful by using techniques like method chaining, as described in ELLS 3.2.
Examples:

```
palindrome?("A man, a plan, a canal -- Panama")  #=> true
palindrome?("Madam, I'm Adam!")  # => true
palindrome?("Abracadabra")  # => false (nil is also ok)

def palindrome?(string)
 # your code here
end
```

(b) Given a string of input, return a hash whose keys are words in the string and whose values are the number of times each word appears. Don't use for-loops. (But iterators like `each` are permitted.) Nonwords should be ignored. Case shouldn't matter. A word is defined as a string of characters between word boundaries. (Hint: the sequence `\b` in a Ruby regexp means "word

boundary".)
Example:

```
count_words("A man, a plan, a canal -- Panama")
# => {'a' => 3, 'man' => 1, 'canal' => 1, 'panama' => 1, 'plan' => 1}
count_words "Doo bee doo bee doo"  # => {'doo' => 3, 'bee' => 2}

def count_words(string)
 # your code here
end
```

Part 2: Rock-Paper-Scissors

In a game of rock-paper-scissors, each player chooses to play Rock (R), Paper (P), or Scissors (S). The rules are: Rock beats Scissors, Scissors beats Paper, but Paper beats Rock.

A rock-paper-scissors game is encoded as a list, where the elements are 2-element lists that encode a player's name and a player's strategy.

```
[ [ "Kristen", "P" ], [ "Pam", "S" ] ]
# => returns the list ["Pam", "S"] wins since S>P
```

(a) Write a method `rps_game_winner` that takes a two-element list and behaves as follows:
- If the number of players is not equal to 2, raise WrongNumberOfPlayersError
- If either player's strategy is something other than "R", "P" or "S" (case-insensitive), raise NoSuchStrategyError
- Otherwise, return the name and strategy of the winning player. If both players use the same strategy, the first player is the winner.

We'll get you started:

```
class WrongNumberOfPlayersError < StandardError ; end
class NoSuchStrategyError < StandardError ; end

def rps_game_winner(game)
 raise WrongNumberOfPlayersError unless game.length == 2
 # your code here
end
```

(b) A rock, paper, scissors tournament is encoded as a bracketed array of games - that is, each element can be considered its own tournament.

```
[
[
  [ ["Kristen", "P"], ["Dave", "S"] ],
  [ ["Richard", "R"],  ["Michael", "S"] ],
],
```

```
[
  [ ["Allen", "S"], ["Omer", "P"] ],
  [ ["David E.", "R"], ["Richard X.", "P"] ]
]
]
```

Under this scenario, Dave would beat Kristen (S>P), Richard would beat Michael (R>S), and then Dave and Richard would play (Richard wins since R>S); similarly, Allen would beat Omer, Richard X would beat David E., and Allen and Richard X. would play (Allen wins since S>P); and finally Richard would beat Allen since R>S, that is, continue until there is only a single winner.

- Write a method `rps_tournament_winner` that takes a tournament encoded as a bracketed array and returns the winner (for the above example, it should return `["Richard", "R"]`).
- Tournaments can be nested arbitrarily deep, i.e., it may require multiple rounds to get to a single winner. You can assume that the initial array is well formed (that is, there are 2^n players, and each one participates in exactly one match per round).

Part 3: anagrams

An anagram is a word obtained by rearranging the letters of another word. For example, "rats", "tars" and "star" are an anagram group because they are made up of the same letters.

Given an array of strings, write a method that groups them into anagram groups and returns the array of groups. Case doesn't matter in classifying string as anagrams (but case should be preserved in the output), and the order of the anagrams in the groups doesn't matter.

Example:

```
# input: ['cars', 'for', 'potatoes', 'racs', 'four','scar', 'creams',
'scream']
#  => output:  [["cars", "racs", "scar"], ["four"], ["for"], ["potatoes"],
["creams", "scream"]]
# HINT: you can quickly tell if two words are anagrams by sorting their
#  letters, keeping in mind that upper vs lowercase doesn't matter

def combine_anagrams(words)
 #   <YOUR CODE HERE>
end
```

Part 4: Basic OOP

(a) Create a class Dessert with getters and setters for name and calories. Define instance methods healthy?, which returns true if a dessert has less than 200 calories, and delicious?, which returns true for all desserts.

(b) Create a class JellyBean that extends Dessert, and add a getter and setter for flavor. Modify delicious? to return false if the flavor is black licorice (but delicious? should still return true for all other flavors and for all non-JellyBean desserts).