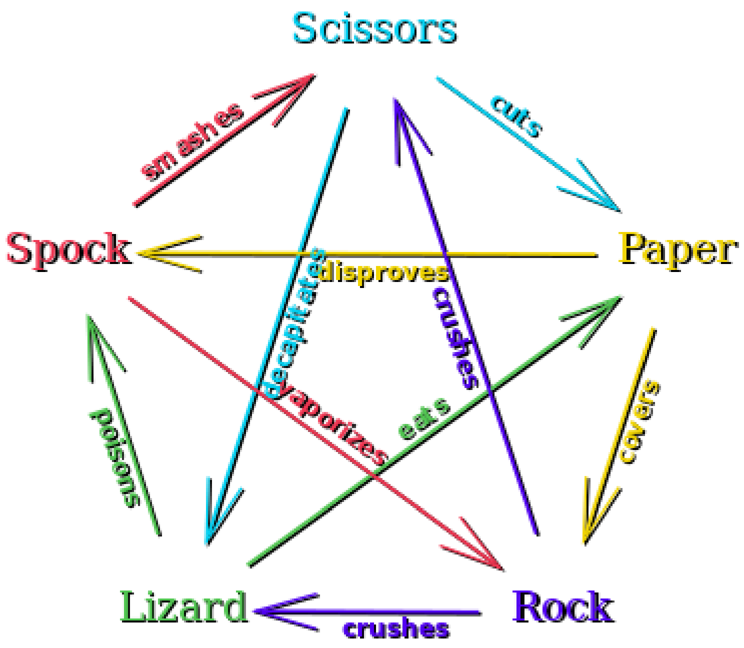
Assignment 7 – RPSLSp

Background

This game is a variant on Rock, Paper, Scissors ([wikipedia article]()). There are 10 possible pairings for the 5-option version of the game. For each item "thrown" there are two things that can defeat it and two that are defeated by it. The phrases used are:



- Scissors cuts paper
- Paper covers rock
- Rock crushes lizard
- Lizard poisons Spock
- Spock smashes scissors
- Scissors decapitates lizard
- Lizard eats paper
- Paper disproves Spock
- Spock vaporizes rock
- Rock crushes scissors

In this assignment you read an input file (`battles3.txt`, or `battles5.txt`) containing a list of the possible battles in a specified format. Your goal is to parse this file and create a two dimensional array of possible outcomes. You will then allow a user to play, choosing which item to "throw" while the computer chooses randomly. The results of the battles will be displayed after each one. A win/loss tally and other statistics will be kept during the game and displayed when the game is concluded.

Program Behavior

Your program starts with an introduction of your choosing. Print anything you like, but keep it to a reasonable number of lines, no offensive remarks, do not read from a Scanner, have no infinite loops, etc.

The program will then print a specified "menu" of options for the user to choose (see the expected output for the exact verbiage), and game will prompt the user for a choice. The user chooses what to play based on the numbered menu, and the program will handle this input robustly. In the example below, the user chose 3 corresponding to *scissors*. The computer randomly chooses a number, and in this case it also chose *scissors* as well. Because they tied, the computer prints:

User (scissors) ties Computer (scissors)

```
<your introduction here>

1. rock
2. paper
3. scissors
4. lizard
5. Spock
Choose your weapon (1-5): 3
User (scissors) ties Computer (scissors)
Battle again (yes/no)? y

1. rock
2. paper
3. scissors
4. lizard
5. Spock
Choose your weapon (1-5): 3
User (scissors) cuts Computer (paper)
Battle again (yes/no)? y

1. rock
2. paper
3. scissors
4. lizard
5. Spock
Choose your weapon (1-5): 1
Computer (Spock) vaporizes User (rock)
Battle again (yes/no)? y

1. rock
2. paper
3. scissors
4. lizard
5. Spock
Choose your weapon (1-5): 5
User (Spock) vaporizes Computer (rock)
Battle again (yes/no)? n

             rock    paper scissors   lizard    Spock
         -------- -------- -------- -------- --------
Computer        1        1        1        0        1
    User        1        0        2        0        1

The computer won 1 time,
the user won 2 times,
and they tied 1 time.
```

Samples of all of the possible phrases used for each battle outcome in the game are given in the file `25phrases.txt`. The actual data for these phrases can be gleaned from a file. Note that you do **not** need to read in `25phrases.txt`. That file is there ONLY for you to see what the exact text of each and every "phrase" that prints out after a battle is.

The files we've provided are

`battles3.txt`
`battles5.txt`
`battles15.txt`
(although we're not sure `battles15.txt` is 100% correct)

The program then asks whether or not to continue the game. If the response starts with Y or y, the game will continue, otherwise it should not keep going (blank strings are not a 'y' or 'Y').

Once the game has ended the program should print a table of the following statistics: the win/loss and how many times the computer and the user chose each of the throws, for example, rock, paper, scissors, lizard or Spock). Notice the singular vs plural cases for the word `time`.

Alternative Input

This program should be able to accept alternative input files, ones that contain data for your standard Rock, Paper, Scissors game or for RPS 15, or 25, or even an input file of your own making. Details of the structure of these files is listed in the Internal Requirements section below

Learning Objectives

- Practice previous concepts
- Demonstrate proper use of two dimensional arrays in Java, including allocating and traversing the contents.
- Demonstrate understanding of reference semantics, particularly as it applies to passing arrays as parameters to methods

External Requirements

Part of your program's score will come from its "external correctness." External correctness measures whether the output matches <u>exactly</u> what is expected. We are very picky about the output matching exactly. Every character and space must match. Use the **output comparison tool** to ensure that your output matches exactly. Programs that do not compile will receive no external correctness points. The general requirements for external correctness are detailed below, specific examples are available on the course website and through the Lakeside Output Comparison Tool.

- You are to exactly reproduce the appropriate console output based on the user's input. There are three main sections to the console output:
  - The introduction
  - The printing of the menu
  - The responding to user input and displaying the results of the battle
  - Continuing to play as long as the user wants
  - Printing statistics at the end.
- There are many potential areas for errors in this program.
  - The battles file may not be present. Here you should reprompt the user for a new file.
  - The battles file may be "malformed" – it may be missing information or have too much information. See the course website for specific error messages to show to the user and what to do if the file is incorrect.
  - This program must be robust in terms of accepting good user input. The following example shows what error messages should be displayed to the user in case of poor user input.

    In any of these cases, the program should not have a runtime error (`Exception`) if the file is not available or if the user enters poor data for their choice.

```
1. rock
2. paper
3. scissors
4. lizard
5. Spock
Choose your weapon (1-5): a
Input is not valid, you need to enter a number.
Choose your weapon (1-5): 0
Input is not valid, you need to enter a number between 1 and 5.
Choose your weapon (1-5): 6
Input is not valid, you need to enter a number between 1 and 5.
Choose your weapon (1-5): 1
User (rock) crushes Computer (scissors)
```

Internal Requirements

Internal correctness means that your program uses the programming elements and structure that are detailed in the list of requirements below. The first few requirements you have seen in other assignments:

- Write a Java class named `RPSLSp` in a file named `RPSLSp.java`. Use exactly this file name, including identical capitalization.
- Please be sure to write all import statements as `import java.<whatever>.*;` for use with GradeIt!
- Be sure to create only one `Scanner` on the console (for user input) in your program. Create this scanner in main and pass it to other methods.
- You may not assume that the y/n answers are all one word responses.
- You may assume that the .java class files and all input or output text files exist in the same directory. Do NOT use absolute path names in your solution.
- A file that does not exist can either be detected by a try/catch block or by using methods that exist for `File` objects.
- You will use the methods you developed for Assignment 6 in `MyUtils.java` that robustly open a file for reading and robustly open a file for writing. You will be turning in `MyUtils.java` along with the `RPSLSp.java` file.
    - You can test whether your method that opens a file for reading is robust by trying to open a file that does not exist. To test if writing to a file is robust, set the permissions on your file to "Read Only" in your operating system, then run your program.
- Your program must use a `Scanner` to process the input file text.
- Produce repetition using loops. Do NOT reproduce repetition using recursion, in other words a method that calls itself, or a pair of methods A and B where A calls B and B calls A, creating a cycle of calls.
- You must define a **`String` class variable** (can be final or not) which contains the name of the file containing the battles (`battles3.txt` or `battles5.txt`) . We will be changing this programmatically through GradeIt.
- The `battles*.txt` files will contain the battle information in the following structure:

The first line of the file will contain how many choices in this game. For `battles5.txt` that would be *5*.

The next line contains all “throw” possibilities, separated by spaces. In this case it will read

rock paper scissors lizard Spock

For simplicity, all “throw” possibilities must be one individual token – meaning the name of a throw may not have any white space. Two word throws (such as “video game”) should be represented as one token such as “video-game” in the file, but displayed to the user without the ‘-‘. This can be one with the `String` method `replace` that is also used in the `MadLibs` programs.

The rest of the file contains all of the combinations of throws and their outcomes. In RPSLSp, there will be 25 lines that look like this

user:computer winner verb

In this example:

paper:lizard lizard eats

Note that battles do not need to be in any order but all of the possible battles (number^2) must be present in order for the file to be a valid input file.

The main purpose of this assignment is to demonstrate your understanding of arrays, how to store and retrieve information in and out of an array and array traversals using `for` loops.

- We recommend at least four arrays:
  - You should use a 1D array to keep the list of possible choices. The index of the array will be very helpful for both the menu choices and for helping you find the battle outcomes in the table described below.
  - For the battle outcomes, create a two dimensional array with the rows being the "user's choice" and the columns being the "computer's choice." The entries in this table is whether the USER or COMPUTER won (or there was a TIE).
  - A second 2D array can hold the "verb" associated with the battle.
  - Other arrays (1 or 2D) are required for keeping track of the win/loss/tie record. Remember - arrays can be passed as reference and modified in the called methods.
- You will likely want to create helper methods to translate from the text ("rock") to a numeric index into your arrays and back.
- You will reuse the methods in your `MyUtils.java` class to help with robust user

  input.
  - For the computer’s choice you should use the `nextInt()` method on a `Random()` object for proper execution in Gradeit. For the diff tool output, we seeded Random with the number 5.
  - As with the guessing game it may be useful to print out the computer's choice - to check that you are getting the right battle responses. You can also "hard code" the computer to always respond with a certain pick - for testing purposes only (make sure to fix this to produce a random value and comment you’re your print statement before submitting).
  - To get an exact match for the output table, here are the `printf` statements used in the code to generate the output:

```
System.out.printf("         %8s %8s %8s %8s %8s\n", "rock”, "paper", …);
System.out.println("         -------- -------- -------- -------- --------");
System.out.printf("%8s %8d %8d %8d %8d %8d\n", “Computer”, ...);
System.out.printf("%8s %8d %8d %8d %8d %8d\n", “User”, ...);
```

    Although to have this work in a more generic way (for RSP3, RSP15 or RPS25), you would have to use a loop with printf statements. HINT. The canonical program uses loops.

- Since arrays are a key component of this assignment, part of your grade comes from using arrays properly. Examples:
  - You should reduce redundancy as appropriate by using **traversals** over arrays (`for` loops over the array's elements). This is preferable to writing out a separate statement for each array element (a statement for element `[0]`, then another for `[1]`, then for `[2]`, etc.).
  - Carefully consider how arrays should be passed as parameters and/or returned from methods as you are decomposing your program. Recall that arrays use *reference semantics* when passed as parameters, meaning that an array passed to a method can be modified by that method and the changes will be seen by the caller. The textbook's case study at the end of Chapter 7 is a good example of a larger program with methods that pass arrays as parameters.
- We will grade your method structure strictly on this assignment. Use at least **four nontrivial methods** besides `main`.
  - These methods should use parameters and returns, including arrays, as appropriate.
  - The methods should be well-structured and avoid redundancy.
  - Each method should have a single coherent purpose, and no one method should do too large a share of the overall task. (try saying what each method does in one or two short sentences). If a method is too long or incoherent, split it into smaller pieces.
  - Avoid "chaining" many calls together without ever returning to main.
- You are required to have the following particular method in your program:
  - A method to read in the battles.txt file and turn it into the battle array.
  - A method to play an individual battle
  - A method to print the statistics.

- Your `main` method should be a concise summary of the overall program. It is okay for `main` to contain some code such as `println` statements. But `main` should not perform too large a share of the overall work itself, such as examining each character of an input line.
- For this assignment you are limited to the language features in Chapters 1 through 7 of the textbook. In particular, you are not allowed to use recursion or object definitions on this assignment

Style Requirements

- There must be a comment header for your class.
- All class variables and methods must have associated Javadoc comments.
- Tricky code and hard-coded values must be explained with in-line comments.
- Code is neatly indented and spaced.
- Give meaningful names to methods/variables, and follow Java's naming standards.
- Limit the lengths of all lines in your program to fewer than 100 characters.
- We check for redundancy on this assignment. If you have a very similar piece of code that is repeated several times in your program, eliminate the redundancy such as by creating a method, by using `for` loops over the elements of arrays, and/or by factoring `if/else` code as described in section 4.3 of the textbook.
- Remember, we will also deduct for magic numbers, particularly for the indices into the battle or statistics arrays. One suggestion is to define class constants for each index of the user and computer, or wins, losses and ties in your arrays. Remember that each class constant requires its own separate Javadoc statement.

Development Recommendations

Here is a list of good recommendations that they should (but are not required to) follow as they develop their program.

- Recall that you can print any array using the method `Arrays.toString` (this may be helpful for debugging purposes). For example:

  ```
  int[] numbers = {10, 20, 30, 40};
  System.out.println("my data is " + Arrays.toString(numbers));
  // my data is [10, 20, 30, 40]
  ```

- We suggest you start your implementation by first analyzing the battles file and writing out the table/2D array by hand to see how you might represent this programmatically.

- Next, we recommend "hard coding" the values in your battle array instead of reading the information from the file. Once you have the game play working with these hard coded arrays, go back and create a method to read in and process the `battles3.txt` or `battles5.txt` file to create the arrays and tables. Recall to "hard code" a 2D array, your syntax looks like this (below is an example for an identity matrix):

```
int[][] ident =  {{1, 0, 0},
                  {0, 1, 0},
                  {0, 0, 1}};
```

- We also suggest you start by testing the program with the `battles3.txt` file, then move onto the `battles5.txt` file.
- You may reuse code from previous programs (`GuessingGame, MadLibs`, or `Outline`) to accept user input, running the game again, or read in files. This is a good opportunity to revisit code that did not work perfectly correctly and enhance it.
- We recommend temporarily "hard coding" the computer's throw, or printing the computer's throw to the console while you are debugging your program.

Optional Challenges

Note: Only attempt the challenges if you are absolutely sure your program matches the expected output 100%. These are not worth any extra credit, they are only ideas for how you can extend your program in interesting ways. If you do attempt a challenge, please make sure you **comment out, or programmatically "turn off"** the additional work so your output matches the expected output exactly

- Add an Artificial Intelligence (AI) to try to beat the user. If you need to store information into a file for your AI, you need to call that file "ai.txt" - otherwise grade it won't function.
- Create a version that has more than 5 choices (there is a 15 and 25 choice version of this game). Create your own `battles.txt` file that would represent these choices and see if your program still works as intended.
- Add a bar graph of the win/loss records, or some other visualization of the game in a `DrawingPanel`.

Perhaps you can think of other ways to extend it yourself!