A3 War!

Announcements and Clarifications

None

Brief Description

War is a two player card game. It is played with a standard 52 card deck (see our last assignment for a description). The rules of War are listed below. Everyone who has played War knows that it is a particularly boring game, namely because there is no aspect of player strategy (there are no choices to make). Instead of playing the game interactively, the computer will play both players and display the course of the game and the outcome. Your program will also accumulate statistics about games over multiple runs.

Your design should be object-oriented using natural objects in the problem space: Queue, Stack, Card, ... You should not use any Java collection classes in your implementation (if you are unsure whether you have done this or not please ask).

Goals and Requirements

- Implement a stack ADT on top of a resizable array. The array should start with enough space for 100 elements and grow by a factor of two each time more space is needed. You do not need to implement shrinking the array.
- Implement a queue ADT on top of two stacks. Two stacks should be the only fields in this data structure.
- Implement each player's hand using a queue
- Write a simple simulation and calculate some statistics
- Implement your ADTs using generics

Rules:

1. Each player is dealt 26 cards from the shuffled deck face down in front of them, this is each player's hand.
2. Each turn:
    a. Each player turns over the top card from their hand.
    b. If the ranks of the cards are different, the player with the higher rank card takes both cards and places them on the bottom of their hand. Aces are the lowest rank card and Kings are the highest.
    c. If the ranks of the cards are the same a war is declared. Each player places three more cards from the top of their hands face down on the table. Repeat step two until the ranks of the cards flipped over are different. At that point the player with the higher rank card takes all the cards placed on the table since the beginning of the war and places them on the bottom of their hand.

3. Repeat step 2 until one player runs out of cards.
4. If at any point a player runs out of cards they lose.

There is one point for variability between programs. The choice is in the order of the cards are placed on the bottom of each player's hand. That choice is up to you; you will explore modifications to this in the Questions section.

We consider infinite games (games that cycle forever) to be games that last more than 15000 turns. There may be games that end in more that 15000 turns but are not infinite, however they are only represent a small minority of the total amount of games. There are ways to check whether the game is cycling though it is beyond the scope of this project (students looking for a challenge can attempt it).

Description

Your program will have two modes of operation. In the first mode it will run a game of War displaying the turn-by-turn action. In the second mode the program will run a number of games and computing several basic statistics about those games. Your program will determine which mode to run in by examining the single command line argument: if the argument is an integer (as parseInt determines) use the second mode, otherwise the argument is considered a filename for mode 1.

Mode 1: Turn-by-turn

```
wolf(1)% java War output1.txt
wolf(2)% java War 1output.txt
```

In this mode the program will list the cards shown face up each turn. The program should not output anything to the console, instead all the output should be directed to a file specified from the commmandline. A sample output is located here. The number of |'s represent the first player's cards and the number of .'s represent the second player's cards. Try to mimic the output style shown in the sample file. If the file to be used cannot be opened or written to the program should exit and indicate the error.

Mode 2: Simulation

```
wolf(3)% java War 356
```

In this mode the program takes an integer from the user on the commandline and runs the game that many times calculating several statistics:

- Number of games run
- Percent of games won by the first player
- Percent of games that lasted more than 15000 turns (endless games)
- Average number of turns in the games that lasted less than 15000 turns
- Average number of aces in the winning player's original hand
- Average number of kings in the winning player's original hand

- Average number of turns that result in a war

The **only** output of the program should be a list of the above statistics to the console. Note that depending on your choices for how the cards get placed on the bottom the statistics may vary. No sample run is provided to maintain the integrity of your results.

Questions to Answer

Include the answers to the following questions at the end of your README.txt file.

1. Algorithm Analysis:

    a. For each of the following ADT methods give the asymtotic running time for the method (as you implemented it), and whether it can be improved upon.

        i. Stack - pop()

        ii. Queue - enqueue()

        iii. Queue - dequeue()

    b. Suppose starting from an empty queue you enqueue *n* items, then you dequeue *n* items, what is the running time of this procedure? Why? What is the average running time for each item in this procedure (this is also called *amortized cost*)?

2. Simulation Analysis:

    a. For each of the statistics what value do you expect your program to return? Why?

    b. Run the simulation version of your program several times with a large number of trials each time (at least 1000). Copy the output into README.txt. Are your results consistent with each other, how do the results vary as you do more or less trials?

    c. Describe how you implemented placing cards on the bottom of each player's hand.

    d. Describe two alternate ways to have implemented that feature.

    e. Implement both of those ways and repeat your experiment from Question 2.b.

    f. Are these results consistent with each other? Do the games play fundamentally the same?

3. In Linux there is a command called **time**. The **time** command is used to measure the time it takes for another command to run.

```
4. wolf(20)% time java War output1.txt
5. 0.358u 0.396s 0:00.88 84.0%     0+0k 0+0io 1pf+0w
```

    The third number, shown in italics, is the amount of clock time it took for the War program to run, in this case it took 0.88 seconds. If you want to know more about the **time** command use the command **man 1 time**. In general to find out information about a command use **man 1 *command_name***.

    Use the Linux time command to measure the actual running time of your simulation. You should take at least 10 measurements in the range from 1-1000 trials and plot a graph of those points. Submit an electronic copy of the graph with

the rest of your program. Based on the graph what is the running time of your program as a function of number of trials?

6. Suppose we required you to implement the check for infinite games properly, give a brief description in psuedo code of the algorithm you might use (**no program code**).

Commenting and Style

- Your program should be written in a style that makes it easy to read and understand.
- At the beginning of each .java file you should include a description of the class and how it interacts with the other parts of your program.
- You are not required to use javadoc style comments.
- If your code is doing something complex or non-standard please comment that portion heavily.

Handin

Please hand all necessary files into your handin directory in a subdirectory named **War**. Your application class should be called **War.java**. There should be exactly one class declared within each .java file. If your program does anything strange (bugs), awesome(extra features) or has a non-intuitive interface please include a file called README.txt which explain them. If there are bugs in your program but you do not describe them in your README you will lose more credit than if you had described them.

Hints

- Develop incrementally
    - Stack ADT
    - Queue ADT
    - War mode 1 (with console output)
    - War mode 1 (with file output)
    - War mode 2 (without statistics)
    - War mode 2 (with statistics)
- Initially choose a simple scheme for ordering the card on the bottom of the hand, also make sure your program works before you go on to do the modifications discussed in the Questions.
- Leverage the code you wrote for assignment 2 (or the code that we provided in the sample solution).
- There are several ways to deal with the different output modes. Here are two you probably not thought of
    - There is a special file on Linux machines called **/dev/null**, you can write anything you want to this file and it just disappears (this will not work on Windows machines).

    - The Java System class has a method called setOut, this method allows you to change where the stream System.out points. You can redirect output from the console to a file or elsewhere. Be sure to save a copy of the original System.out so that you can restore console output if you need to. Look in the Javadocs for System for more information.

- If you are unfamilar with using generics look at the sample solution for assignment 2 and section 2.5.2 in your text.

- Running a 1000 trial simulation should take no more than two minutes on a lab machine (in fact it should take much less time), if it takes longer you're probably doing something very inefficiently, look at your data structures and analyze the running time of your methods.

- If you're having trouble figuring out how to implement a queue using two stacks, consider the following hint: when you enqueue push onto the first stack, when you dequeue pop off the second stack.

- The results for Question 2 will vary depending on how you implement the ordering, just because your result is different from someone else's doesn't mean that either of your programs are wrong.