Homework #1

RELEASE DATE: 03/19/2021

DUE DATE: 04/16/2021, BEFORE 10:00 on github classroom

QUESTIONS ARE WELCOMED ON THE NTU COOL FORUM.

You need to upload your submission files to the github repository under the exact guidelines that will be provided on NTU COOL.

Any form of cheating, lying, or plagiarism will not be tolerated. Students can get zero scores and/or fail the class and/or be kicked out of school and/or receive other punishments for those kinds of misconducts.

Discussions on course materials and homework solutions are encouraged. But you should write the final solutions alone and understand them fully. Books, notes, and Internet resources can be consulted, but not copied from.

Since everyone needs to write the final solutions alone, there is absolutely no need to lend your homework solutions and/or source codes to your classmates at any time. In order to maximize the level of fairness in this class, lending and borrowing homework solutions are both regarded as dishonest behaviors and will be punished according to the honesty policy.

You should write your solutions in English. We do not accept solutions written in any other languages.

This homework set comes with 200 points and 20 bonus points. In general, every homework set would come with a full credit of 200 points, with some possible bonus points.

Overview

You are going to develop a poke card game called Big-2. Before you actually start extend your game to more sophisticated rules and human/bot players, you need to make sure that you can simulate the game correctly with a small set of rules.

The game is called Big-2 because the highest card you can play is a 2. Nothing beats a 2. Some of you may have played this game before, but please note that we are not asking you to implement your understanding of the game. We are asking you to implement *the set of rules defined below* and think about how to extend them in the future.

The Big-2 game is played with a standard deck of 52 cards and four players. The cards are ordered by certain rules; a player is allowed to release some cards in her/his turn only if her/his cards are in a higher order than the topmost ones on the table. Otherwise the player needs to "pass." The first player who empties her/his all hand cards wins, and the game immediately ends up.

You must follow the self-contained spec below very carefully regarding the rules. There are many variations and implementations of the Big-2 game on the internet, but those are generally more distracting than helping. Given that our spec is self-contained, you are encouraged to simply read and follow the spec in this homework, rather than getting lost in the (noodle-oriented) code that you can find online.

Glossary

1. Big-2: the poke card game we are going to develop
2. player: an entity that plays the game
3. rank: one of $\{3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, A, 2\}$
4. suit: one of {club ♣, diamond ♢, heart ♡, spade ♠ }
5. card: a combination of (rank, suit)
6. deck: a stack of 52 different cards
7. deal: the action of a player getting a card from the deck

8. hand cards: the cards owned by a player
9. card pattern: a kind of card combinations
10. play: the action of releasing a legal card pattern from a player's hand cards to the table in a player's turn
11. round: The game consists of several rounds (tricks). In each round, players take turns to play until three passes occur in a row.
12. top play: the highest-order card pattern in the round
13. top player: the player who played the top play

Game Flow

1. The game consists of four players. When the game starts, get the name of each of the player. The player will be numbered $\{0, 1, 2, 3\}$ for simplicity, where player $(i + 1)\%4$ is called *the next player* to player i
2. Get a shuffled deck of 52 different cards and deal them to the four players one by one until the deck is empty.
3. In the first round of the game:
   [a] The player who owns a ♣3 plays a legal card pattern that contains ♣3 to start the first round, and the card pattern naturally becomes the top play.
   [b] Then, the next player either
   - chooses to `PASS` and does not play any cards, or
   - plays a legal card pattern of a higher order than the top play, and the new card pattern becomes the top play

   [c] Repeat the previous step on the next player, and so on, until three consecutive `PASS` have been received.
4. Start the next round of the game, with the same rules as the first round, except:
   [a] The player who was the last top player in the previous round plays a legal card pattern to start the round.
5. Repeat for several rounds until one player has an empty set of hand cards in the middle of any round, and declare the player winning *immediately*.

Card Patterns and Ordering

There are four card patterns below. Only cards of the same card pattern can be compared.

- single: a single card, with ranks ordered by $3 < 4 < 5 < \ldots < 10 < J < Q < K < A < 2$, and suits ordered by ♣ < ♢ < ♡ < ♠. That is, ♠2 is the highest-order single card, and ♣3 is the lowest-order one.
- pair: two cards with the same rank; e.g., J-J, 3-3, where the highest-order single card within each pair is taken to compare the rank of two pairs.
- straight: any five cards in a sequence; e.g., 3-4-5-6-7, 10-J-Q-K-A, J-Q-K-A-2, or K-A-2-3-4, where the highest-order single card within each straight is taken to compare the order of two straights. *Note that this may be significantly different from some other variants. For instance, for Q-K-A-2-3 and K-A-2-3-4, the "2" in each straight will be taken as the comparison card.*
- full house: five cards that include three cards of the same rank (triple), and two cards of another rank (pair); e.g., 3-3-3-2-2, A-A-A-7-7, where the rank of the triple within each full house is taken to compare the order of two full houses.

Input Format

```
<Shuffled Deck>
<Player 0's name>
<Player 1's name>
<Player 2's name>
<Player 3's name>
<intended action 1>
<intended action 2>
...
```

The first line specifies the cards in the shuffled deck.

```
<Shuffled Deck> := <suit>[<rank>]  <suit>[<rank>] ... <suit>[<rank>]
```

You can assume that the shuffled deck contains 52 distinct cards. `<suit>` is one of `C`, `D`, `H`, `S` representing ♣, ♢, ♡, ♠, respectively; `<rank>` is one of `3`, `4`, ..., `9`, `10`, `J`, `Q`, `K`, `A`.

For the shuffled deck, the leftmost card is placed at the bottom of the deck and the rightmost card is placed at the top. During dealing, a card from the top of the deck is given to one player, in the order of player 0, player 1, player 2, player 3, player 0, player 1, and so on.

Each of the next four lines contains a player's name, which contains alphanumeric characters (A-Z+a-z+0-9, case-sensitive). Then, each `intended action i` line contains either -1 to represent `PASS`, or a sequence of non-negative integers, each separated by a single space, to represent the choice of cards to play from the players' hand cards.

In the beginning of the first round, the player of the intended action is the one that carries `C[3]`; in the beginning of the next rounds, the player of the intended action is the one that was the last top player in the previous round. Every player keeps trying an intended action, which can be either legal or illegal, until a legal action can be performed. Then, the next player becomes the player for the next intended action.

Output Format

- In the beginning of each round, output a line of

  ```
  New round begins.
  ```

- At every player's turn, first output a line of

  ```
  Next turn: <player's name>
  ```

  Then, print out the player's hand cards in a pretty format as follows

  ```
  0    1    2    3    4    5    6     7    8  ...
  S[3] H[4] S[4] H[6] H[7] C[8] C[10] S[K] C[2]  ...
  ```

  The hand cards should be *ordered* from the lowest-order to the highest-order. In the second line of the pretty format, each card is separated by a single space, and in the first line, the index of the card is strictly left-aligned to each card's suit.

- Then, if the player plays a legal card pattern, print out a single line of

  ```
  Player <player's name> plays a <card pattern> <suit>[<rank>] <suit>[<rank>] ...
  ```

  where `<card pattern>` is one of `single`, `pair`, `straight`, or `full house`, and all cards in the card pattern ordered from the lowest to the highest. Otherwise, print out a single line of

  ```
  Invalid play, please try again.
  ```

- If the player passes legally, print out a single line of

`Player <player's name> passes.`

Otherwise, print out a single line of

`You can't pass in the new round.`

if the player is at the beginning of each round (which should be the only place where passing is illegal.)

- When the game is over, print a single line of

  `Game over, the winner is <winning player's name>.`

Java Programming

- Java SE Version 11 will be used for grading—you can consider an open JDK here `https://adoptopenjdk.net/`.
- We will use `java Main` to run your program. That is, the entry point of your program should be the `main` static function inside the `Main` class.
- We will use `javac -sourcepath src/` to compile your program. That is, the source files should be put under the `src/` directory.
- inputting: you are suggested to use `new Scanner(System.in)` to instantiate a `Scanner`, and use the scanner's methods like `next`, `nextInt`, or `nextLine` for consuming and parsing the input from the standard input stream.
- outputting: you are suggested to use `System.out.print`, `System.out.println`, or `System.out.printf` for outputting.

Grading Criteria

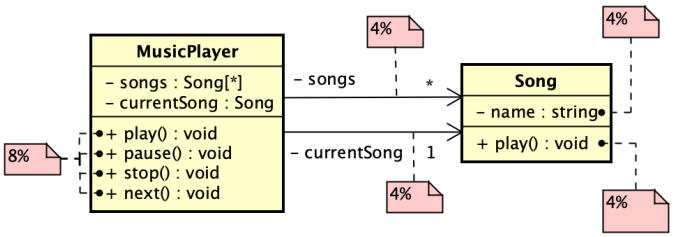
- program correctness (each 20%):
  - public test case 1: always-play-first-card (single)
  - public test case 2: normal-no-error (single + pair)
  - public test case 3: normal-no-error (single + pair)
  - public test case 4: illegal actions (single + pair)
  - public test case 5: anything (single + pair + straight)
  - public test case 6: anything (single + pair + straight + full house)
  - private test case 7: anything (single + pair + straight + full house)
  - private test case 8: anything (single + pair + straight + full house)
- software design (40%)
- bonus software design (bonus 20%): Open-Closed Principle (OCP), which states "*Your software should be open for extension, but closed for modification.*" Discuss how you can extend your program by only writing new classes and modifying only `Main` (which is commonly called the client class), but not any other classes, to extend the Big-2 game with new card patterns, such as a triple (3 cards with the same rank).

Design Report

The purpose of the design report is to help the TAs grade the software design (and bonus) part by human. We hope that you can illustrate every class of your design with

- its name
- a one-sentence description of its duty
- a short description of how it possibly interacts with other classes

The TAs will try to grade by checking whether your design satisfies some "ideal" design choices using an internal graph of class relations like this.



We are not strictly asking you to produce this kind of graph (as we have not taught you about how to do so). But please feel free to illustrate your classes with a similar graph if you want to.

Other than those required above, please feel free to add anything that helps the TAs understand your design. But **please keep your design report as concise as possible**.

While the TAs grade with the "ideal" design choices, there is always a possibility that your design choices are better than the "ideal" ones. In this case, you can certainly check/argue with the TAs to get the points that you deserve afterwards. So please do not be too worried about whether your current design choices are "ideal" or not.

Submission Files

- your source code in the `src/` directory of your repository.
- a design report `Design-Report.md` in the root directory of your repository.

Tips from TAs

1. Keep every Java class less than 150 SLOC (source lines of code). If there is any class that has greater than 150 SLOC, it just means the class does too many jobs, which violates the Single Responsibility Principle
2. Model the Big-2 game flow and every rule very carefully! You absolutely do not want to waste your time on trial-and-error over our test cases! Drawing a flow-chart may help you capture all the details.
3. Separate the card patterns' evaluation algorithms from the game flow.
4. Separate your ordering rules from the game flow.
5. Separate your players' trial-and-error procedure from the game flow.
6. Use the `enum` type for suit and rank, instead of unnecessary primitive types.
7. Put the game flow in a separate class, instead of your `Main` class.