Preamble

Creation of a workspace for the exercise

Create a new directory `CSIE_340679`, from your operating system's file browser, or with the command `mkdir ~/CSIE_340679` in a terminal. This directory will be your *workspace*, it will contain all the course exercises.

Creation of a project dedicated to HW1

1. Launch *Visual Studio Code* (*VS Code* hereafter), for example by running the code command in the terminal, or by clicking on a shortcut; the `[Alt]` key makes the horizontal menu bar appear/hide.
2. In *VS Code* type the key combination `[Ctrl]+[Shift]+[P]` (or `[Alt]+[V]` then click on *Command Palette*) then type "`java:create`" in the bar that appears and select Create Java Project and the No build tools option .
3. A window opens and prompts you to choose your workspace , that is, the `CSIE_340679` directory that you created by following the instructions in the previous paragraph.
4. Enter the name of your project (`HW1`) in the text bar.

Source recovery

Left click on the src.zip link , then click on "Extract", finally select your project directory (i.e. the HW1 directory which is inside the `CSIE_340679` directory) when the unzip tool asks you to and click on "Extract" again.

Enable assert in your Java virtual machine

In *Visual Studio Code*, type the key combination `[Ctrl]+[,]` (the second key is 'comma'), enter *vm args* in the search bar. The item `Java>Debug>settings: Vm Args` appears with a text bar in which you must write `-ea` (for enable asserts).

Documentation

To obtain a description of a standard Java class, use a search engine and enter `Java` followed by the name of the class for which you want documentation (eg `Java LinkedList` or `Java Array`).

The official (internet) documentation is here: `Java 11`.

Warning: All your code must be written in the `HW1.java` file

In order to minimize the risk of handling errors when uploading files, all the classes that you have to modify are grouped together in the same `HW1.java` file that you must upload after each question.

(We remind you, however, that except in exceptional circumstances, we tend to write different classes in different files, both for the readability of the code and for the efficiency of compilation.)

The Battle Game

War is a card game between two players. At the beginning of each game, the 52 cards in the game are randomly divided into two decks of the same size, each assigned to the opponent. The outcome of a game is entirely determined by the initial distribution of the cards: the goal of this HW is to write a program that shuffles the cards, simulates the progress of the game, and indicates the winner (a “draw” is possible).

1 Card decks

A standard deck of playing cards contains 52 cards, each characterized by

- its color $\in \{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\}$ and
- its value $\in \{1 < \cdots < 10 <$ Jack $<$ Queen $<$ King$\}$.

In the following, `nbVals` refers to the number of different values in the deck of cards, which is 13 in principle, but we leave ourselves the option of using more or less in the test. On the other hand, we commit to always using 4 colors. The deck therefore contains 4 cards of each value, or `4*nbVals` cards in total. By convention, we will number the values from `1` to `noVals`.

The color of the cards is not taken into account in the game of Battle. A *deck* of cards is therefore a row of integers (we take from above, we add from below) whose elements are all between `1` and `nbVals` and in which the same integer appears at most 4 times. A deck therefore contains at most `4*nbVals` cards.

A deck of cards is represented by a linked list of integers (i.e. `LinkedList<Integer>`) encapsulated in the `Deck` class which already contains:

- a field `LinkedList<Integer> cards`,
- the following manufacturers:
  - `Deck()` : an empty deck,
  - `Deck(LinkedList<Integer> cards)` : a deck whose field `cards` is given,
  - `Deck(int nbVals)` : all cards arranged in ascending order,
- a method `Deck copy()` that returns a copy of the deck,
- a method `String toString()` that returns a string representing the deck.

In the class `Deck`, complete the following methods:

- `int pick(Deck d)` : if deck d is not empty, removes its first card, adds it to the end of the current deck (`this`), and returns its value. Otherwise, returns `-1`, while the decks remain unchanged;

- `void pickAll(Deck d)` : removes all cards from deck `d` one by one and adds them to the end of the current deck;
- `boolean isValid(int nbVals)` : returns `true` if the current deck is a valid deck, i.e. if it does not contain integers outside of $\{1, \ldots, \texttt{nbVals}\}$ and no more than 4 cards of the same value.

Test your code by running `Test1.java`.

2 The American mix

The *American shuffle*, or *riffle shuffle* in English, is a technique commonly used to shuffle cards (i.e. randomly permute the elements of a deck):

1. The deck of cards is arbitrarily cut into two piles (which are wedged between the thumb and index finger of each hand),
2. then we drop the cards one by one on the table so as to interlace them (this is the riffle).

The manipulation is illustrated by the video tutorial and explained in detail in this Wikipedia page. We will implement in the class `Deck` a method that performs this mixing.

2.1 The cut

The cutting of the deck into two sub-decks is done randomly approximately in the middle of the pile of cards. We will therefore prefer a random draw following the binomial distribution rather than the uniform distribution. The binomial distribution of parameter n (in this exercise, n is the size of the deck of cards) is obtained by counting the number of "heads" draws obtained on n tosses of a fair coin (each toss returns "heads or tails"). The method `double Math.random()`, which returns a value chosen at random in the interval $[0, 1[$ can be used to simulate such a toss.

In the Deck class, complete the method `int cut()` so that it returns the number of cards in the "first" deck, drawn randomly following the binomial distribution.

Test your code by running `Test21a.java`. The latter calculates the deviation in "sup norm" between the theoretical distribution of the binomial law (of parameter n) and that which is obtained empirically from `m` calls to the method `cut` (on a set of n cards). For `n = 52` and `m = 100000` the deviation rarely exceeds `0.0025` .

In the class `Deck`, complete the method `Deck split()` so that it removes and returns the first `c` cards from the deck, with the value `c` given by a call to the method `cut`.

Test your code by running `Test21b.java`. This program cuts a deck of 52 cards (about a hundred times) and checks that the two resulting decks give the full deck.

2.2 The Riffle

In *riffle*, the next card to fall is the first card in one of the two piles produced by the cut. The probability that it falls in the first is decided to be `a/(a+b)` where `a` and `b` denote the respective numbers of cards in the two piles, which reflects fairly well what happens in practice. Note that the parameters `a` and `b` must be updated each time a card falls.

In the class `Deck`, complete the method `void riffleWith(Deck d)` so that it merges the cards in deck `d` with those in the current deck. As a result, deck `d` is empty and the result of the merge is in the current deck.

Hint : create a third deck `f` , initially empty, from which we call the method `pick` until the decks `d` and `this` are exhausted . At the end of this step, the field `f.cards` contains the result of the *riffle*. All that remains is to update the field `this.cards`.

Test your code by running `Test22.java`, which does:

- 10000 times the *riffle* of decks $1, 2, \ldots, 26$ and $27, 28, \ldots, 52$, checking each time that the result contains each of the two decks in order, and
- 10000 times the *riffle* of the decks $1, 2, 3, 4, 5$ and $6, 7, 8, 9, 10$ and checks that each of the 252 possible outcomes (252 is the binomial coefficient $C(5, 10)$) appears at least once.

2.3 Combine everything

The distribution of cards after an American shuffle is not uniform since once the cut is made, the order of the cards in each of the two piles is preserved by the *riffle* .

On the other hand, by iterating the process enough times (of the order of $\log n$ times for a deck of n cards) we obtain an (almost) perfectly random distribution, as demonstrated by D. Bayer and P. Diaconis in a now famous article.

In practice, 7 American melds are enough to properly shuffle a deck of 52 cards.

In the class `Deck`, complete the method `void riffleShuffle(int m)` so that it performs m American shuffles (`split` then `riffle`) on the current deck of cards.

Test your code by running the program `Test23.java` . This program repeats the following experiment a million times:

1. create a new deck of 52 cards,
2. shuffle this game via the method `riffleShuffle(7)`,
3. check if the obtained package is suspicious .

Empirically, a package is declared suspicious as soon as it contains at least:

- 14 twins (2 consecutive identical cards), or
- 3 quadruplets (4 consecutive identical cards), or
- 2 octuplets (8 consecutive cards where only 2 values are observed)

Still empirically, out of a million mixes, you rarely get 5 suspect packages. If this happens, repeat the test. If the symptoms persist, consult your code.

Cultural note: The speed of convergence towards the uniform distribution is not constant, however, and in particular there is a threshold beyond which the distribution becomes very uniform, while below it remains very concentrated. This latter property is the keystone of many card tricks. The third section of the article by Diaconis, Graham and Kantor provides more details on this point. Note that Diaconis was himself a famous magician before becoming a probabilist.

3 Simulation of a battle game

The rules of the battle are simple: at the beginning of the game, the cards are dealt between the two players, who put them in a pile in front of them (without looking at them). Each turn, each player takes the card on top of their pile. The player with the highest value card wins both cards and places them under their pile. In the event of a tie, it is said that there is a battle and each player then takes two new cards on top of their pile, the first by placing it on the trick without looking at it and the second to compare it to that of the opponent. In the event of a new battle, this process is repeated; otherwise, the winning player takes the trick. The game is interrupted after n tricks (n fixed in advance) or as soon as one of the players has no more cards. The player with the most cards at the end of the game is the winner (there is a tie if the two players have the same number of cards).

3.1 Start of the game

A battle part is modeled by the class `Battle`, which contains:

- the fields `player1` , `player2` , and `trick` of type `Deck` which represent the current state of the game:
  - `player1` : the first player's deck of cards,
  - `player2` : the second player's deck of cards,
  - `trick` : the fold.
- a constructor `Battle()` that constructs an empty battle,
- a constructor `Battle(Deck player1, Deck player2, Deck trick)` that constructs a battle with the given `player1`, `player2`, and `trick` fields,
- a method `Battle copy()` that returns a copy of the battle,
- a method `String toString()` that returns a string representing the battle.

Complete the constructor `Battle(int nbVals)` so that it initializes a battle part, in other words this constructor must:

- create a new deck of `4*nbVals` cards (with the constructor `Deck(int nbVals)`)
- shuffle this deck (with the method `riffleShuffle(7)`)
- distribute this deck to the two players (giving one card to each player alternately until the deck is used up).

Test your code by running `Test31.java`.

3.2 Progress of a fold

In the Battle class , complete the method `boolean isOver()` so that it returns `true` if one of the players has no more cards and `false` otherwise.

In the class `Battle`, complete the method `boolean oneRound()` so that it simulates a fold:

1. each player draws a card and puts it in the trick (first `player1` , then `player2`) ,
2. the player who turned over the strongest card wins the trick, which he adds to the end of his deck without changing the order of the cards in this trick,
3. in case of a tie there is a "battle":
    - each player draws a first card and adds it to the trick (these cards are not compared; it is at this time that high-value cards can change hands)
    - we return to step 1

The value returned by the method indicates whether the fold was successful:

- if at a stage of the trick, a player must draw a card while he no longer has any in hand, then neither player plays a card, and the method immediately returns `false` (i.e. the game ends);
- if both players have enough cards to complete the trick, the method returns `true` (i.e. the game continues).

NB : The method can return `true` even if one of the players has no cards left in hand, for example if he lost the trick by turning over his last card. Similarly, it is possible that both players are out of cards (this happens when both players have the same hand at the start of the trick).

Test your code by running `Test32.java`.

3.3 Game progress

In the class `Battle`, complete the method `int winner()` so that it returns `1` (respectively `2`) when the first (respectively the second) player has strictly more cards in hand than his opponent, or `0` if both players have the same number of cards in hand.

Using the method `winner`, complete the method `int game(int turns)` so that it simulates a game by setting `turns` to the maximum number of tricks played, and which returns `1` or `2` to indicate the winner, or `0` if both players have the same number of cards in hand at the end of the game (i.e. if both players are out of cards or at the end of the last turn).

Test your code by running `Test33.java`.

4 Infinite games (optional but exciting)

So far we have only considered battle games with a finite number of moves fixed in advance. We will lift this limitation.

To do this, we need to add a mechanism to the simulator that detects infinite games. We denote by S_k the state of the game at turn k , that is, the pair of packets held by the two players at turn k .

A game is infinite when the sequence of states $S_0, S_1, \ldots, S_k, \ldots$ is infinite. Since there are only a finite number of possible distinct states and the state S_{k+1} is entirely determined by the state S_k, such a sequence is necessarily periodic from a certain rank.

We will play two games `b1` and `b2` simultaneously. At the beginning of both games, the players have the same cards in hand, but the second game (i.e. `b2`) plays two tricks each time the original game (i.e. `b1`) plays one. In other words, we calculate on the one hand the sequence $S_0, S_1, \ldots, S_i, \ldots$ (original game) and on the other hand the sequence $S_0, S_2, \ldots, S_{2i}, \ldots$ (second game). The game is infinite if and only if there exists an integer i such that $S_i = S_{2i}$: this is the hare and tortoise algorithm, due to Floyd.

4.1 Implementation

Recall that we provided a method `Battle copy()` that returns a clone of an object of class `Battle`. The original and its clone occupy disjoint memory locations, so a modification of one does not affect the other.

To test whether two battles are identical, we will test the equality of the strings representing them with `b1.toString().equals(b2.toString())`.

In the class `Battle`, complete the method `int game()` so that it simultaneously plays a turtle game (which will be played by the clone) and a hare game ie at double speed (which will be played by the original, in other words by `this`) having the same initial state and which returns:

- 0 in case of a draw (i.e. if both players are out of cards),
- 1 or 2 to indicate the winning player,
- 3 in case of infinite game.

If the game ends, `this` must contain its final state.

Test your code by running `Test41.java`.

4.2 Statistics

In the class `Battle`, complete the method `static void stats(int nbVals, int nbGames)` so that it simulates `nbGames` battle games, each with `nbVals` values, at the end of which the number of wins for each player, infinite games, and draws is displayed.

Test your code by running `Test42.java` *several times*. It runs the method `stats` with a deck of 44 cards (11 values), then with a deck of 48 cards (12 values), and finally with a deck of 52 cards (13 values). What do you notice?

With our current implementation, `player1` always puts his card before `player2` to the trick. We'll modify the class `Battle` so that `player1` and `player2` take turns putting their card first to the trick. To do this, add a field `boolean turn` to the class `Battle`, initialized to `true` in the constructors, and modify the method `boolean oneRound()` so that whenever both players have to put a card to the trick, `player1` starts if `turn` is `true` , while `player2` starts if `turn` is `false` , and `turn` changes value. Rerun `Test42.java`.