THE UNIVERSITY OF WESTERN AUSTRALIA

School of Computer Science and Software Engineering

CITS1001 Object-oriented Programming and Software Eng

Document version number: 1.1.3.
Document date: 2018-05-24
(Fixed info on `tryPosition()`)
Check the CITS1001 web-site to ensure that you are aware of the latest version.

Project 2: Playing-card cryptography

Submission deadline: 5pm, Friday 1 June 2018.
Value: 15% of CITS1001.
To be done in pairs. Individual is also fine; groups of more than two are NOT allowed.
Before you submit, make sure that all names and student numbers are listed on the submission.

You should construct a Java program containing your solution to the following problem. You must submit your program electronically using *cssubmit*. No other method of submission is allowed.

You are expected to have read and understood the University's guidelines on academic conduct. In accordance with this policy, you may discuss with other students the general principles required to understand this project, but the work you submit must be the result of your own effort.

You must submit your project before the submission deadline above. The penalty for late submission is described here.

While perusing the library's copy of Donald Knuth's *The Art of Computer Programming* one evening, a sheet of paper between two pages falls out. Looking at it, you discover that it contains an encrypted message, and instructions for a *cryptosystem*, Pontoon, for decrypting it. The cryptosystem is designed so that it can be implemented using a deck of playing cards – but, not having a deck of cards to hand, you decide to implement it in Java.

In this project you will write a Java program that implements the Pontoon cryptosystem, and then use it to decode an encrypted message.

The Pontoon Cryptosystem

You will implement parts of three Java classes – `Deck`, representing a card deck and operations that can be performed on it, `Encoder`, which can be used to encode and decode messages using the cryptosystem, and `Message`, which contains the encrypted message:

- *Deck.java*
- *Encoder.java*
- *Message.java*

Each link above gives you a skeleton of the required class, including instance variables, constructors, and methods. You are required to complete the methods whose bodies are marked with the comment TODO. A suggested order for tackling the classes, and descriptions of what the methods do, follows.

Deck

This class represents a deck of cards and operations that can be performed on it (together with some utility methods, useful in implementing the operations). The Pontoon cryptosystem requires a deck of cards with two jokers that can be distinguished in some way. (In many decks, the image on one of the jokers will be slightly larger or in some way different from the other.)

Cards are represented as integers from 1 through to 54, inclusive; the numbers 1 through to 52 represent cards in the four suits (clubs, diamonds, hearts, and spades), and the numbers 53 and 54 represent the first and second joker. It is an error to try and create a deck where multiple cards have the same value.

A suggested order of implementation is:

- `allDifferent()`
- `findCard()`
- `validateCards()`
- `shiftDownOne`
- `tripleCut()`
- `countCut()`
- `shuffleArray()`

Each method has documentation describing its effect. Some additional information on methods, and examples to illustrate some of them, follow. (Note that you can also look in the test class code, given below, for examples.)

- `shiftDownOne()`. Suppose you were using a deck of only 12 cards: 1,2,3,4,5,6,7,8,9,10,11,12. Then a call to `shiftDownOne(10)` would put the cards into the following order: 1,2,3,4,5,6,7,8,9,11,10,12. Whereas calling `shiftDownOne(12)` on that deck would result in this order: 1,12,2,3,4,5,6,7,8,9,10,11.
- `tripleCut()`. The method takes as its argument two *positions* which divide the deck into three "chunks", and the first and last "chunks" are swapped.

  By way of example: again, suppose you are using a deck of only 12 cards: 1,2,3,4,5,6,7,8,9,10,11,12. A call to `tripleCut(1,9)` on the 12-card deck would result in the new order: 11,12,2,3,4,5,6,7,8,9,10,1. Even if one or more "chunks" consist of zero cards (i.e., are empty), the operation should still work.
- `countCut()`. If `countCut(4)` were applied to the 12-card deck from the previous examples, this would result in the new card order: 5,6,7,8,9,10,11,1,2,3,4,12.
- `shuffleArray()`. When called, this method should shuffle the deck into a new order. There are three JUnit tests for this method, `testShuffle_low_grade`, `testShuffle_medium_grade`, and `testShuffle_high_grade`. They measure the amount of *entropy* (a measure of disorder) your method introduces – you should aim to pass all three tests. The highest entropy is achieved when each possible ordering of the deck has the same probability of occurring when the method is called.

Encoder

The `Encoder` class is used to encrypt or decrypt a message. When created, an `Encoder` is initialized using a `Deck` object with its cards in a particular order (or, if no `Deck` is specified, the `Encoder` class uses a `Deck` with its cards in the default order: 1, 2, 3 … through to 54).

The Pontoon cryptosystem only works on capital letters from the English alphabet. The `sanitize` method is used to remove characters from a String which the system cannot handle.

Internally, Pontoon operates not with letters (`'A'`, `'B'`, `'C'` etc.), but with numbers (`int`s) representing those letters. `'A'` is represented by the number 1, `'B'` by the number 2, and so on. The methods `charToInt` and `intToChar` convert between these two representations.

The `encodeChar` and `decodeChar` methods encode and decode, respectively, a character from an unencrypted or encrypted message. Their first argument is a character from the message; their second argument is a character from the "*keystream*", a sequence of characters generated by the Pontoon system using the deck of cards. (We show how this keystream is generated soon.) The `encodeString` and `decodeString` methods are similar, but operate on an entire message, rather than just a single character. Descriptions of exactly how those methods should operate is given in the source code.

The `nextKeyStreamChar` method generates a *keystream* character, based on the current state of the deck.

The algorithm it uses is as follows:

- locate the first joker, and *shift it down one* position (treating the end of the deck as if it joins to the start).
- locate the second joker, and shift it down *two* positions (treating the ends of the deck as if joined, as before).
- Perform a *triple cut*, using the current positions of jokers 1 and 2.
- Find the value of the card currently at the bottom of the deck. If it has value 54 (i.e., it is the second joker), subtract one from it. Now perform a *count cut* using that value.
- Look at the value of the top card. If it has value 54 (i.e., it is the second joker), subtract one from it. Now use that value as a position in the deck, and see what card is at that position – this is the *output card*.
- If the output card is a joker, we repeat the algorithm. If it isn't, we return the value of the output card, converted into a `char`.

Finally, the `encrypt()` and `decrypt()` methods encrypt or decrypt a string. For each character in the string, they call `nextKeyStreamChar()` to generate a keystream character; then they call `encodeChar()` and `decodeChar()` to encode or decode the character, respectively.

A suggested order to tackle the methods is as follows:

- `sanitize()`
- `charToInt()` and `intToChar()`
- `encodeChar()` and `decodeChar()`
- `encodeString()` and `decodeString()`
- `nextKeyStreamChar()`
- `encrypt()` and `decrypt()`

Message

The `Message` class holds the encrypted message which you are trying to decrypt, as a static `String` variable. It also holds the ordering of the card deck that needs to be used to decrypt the message … except that you notice that the position of one card, the King of Hearts (with card value 39) is

USEFUL LINKS

- Unit outline
- Unit timetable 2018
- Java 8 API
- help1001
- csentry
- cssubmit
- csmarks
- IT help for students
- Compulsory online modules

missing.

You will need to complete the `tryPosition()` method, which takes as a parameter a possible position in the partial deck (from 0 to 53) where the King might go. It returns a String of comma-separated card values, with the King inserted in the appropriate place. (This “deck string” can be used to create a Deck object with its cards in that order.)

You will also need to complete the `Message()` constructor for the `Message` class. See the source code for details of what it should do.

Submission

Submit your three completed files, `Deck.java`, `Encoder.java`, and `Message.java` via *cssubmit*. **Before you submit, make sure that all names and student numbers are listed on the submission.**

Your file names, class names, and method names and signatures must match the specifications *exactly*. The marking process is partly automated and any deviation in the names or signatures will cause problems, which will be penalised. It is ok to add other methods if you like, as long as you do not change the names or signatures of existing methods.

Common mistakes are to submit *.class* files in place of *.java* files, or to submit our test classes in place of your code. If you do one of these, you will be notified as soon as we become aware, but **you will be due for any applicable late penalty**. It is easy to check your submitted files after you have submitted them – do it!

Assessment

Your submission will be assessed on

- completeness: how many of the methods you have written;
- correctness: whether your methods implement the specifications exactly;
- clarity: whether your code is clear to read and well-constructed.

Testing classes are provided for you to validate your program before submission:

- *DeckTest.java*
- *EncoderTest.java*
- *MessageTest.java*

Note however that the correctness of your program is your responsibility. **The testing classes are (intentionally) quite brief, and a program that passes all of the tests is not guaranteed to be free of errors.** "Testing shows the presence, not the absence of bugs." -- E.W. Dijkstra.

Feel free to augment the testing classes with more tests of your own.

In ensuring your code is clear, the CSSE Java style guide may be a useful resource.

Challenge task

In a hollowed-out section of Volume 2 of *The Art of Computer Programming*, you unexpectedly find a deck of cards, and an additional encrypted message. A note informs you that six of the cards are out of place, as they fell out when the volume was being shelved. Unfortunately, it doesn’t tell you which the six cards were. The encrypted message, and the (disordered) sequence of cards is here. Can you work out what the decrypted message is? There will be a prize** for the first person to email Arran a Java program that decrypts the message.

**All prizes are awarded or not solely at Arran's discretion.

Marks breakdown

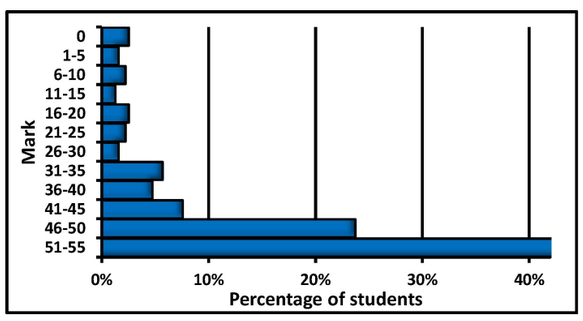
There were 45 marks for correctness, distributed as follows:

- Deck class:
  - `allDifferent` – 2
  - `validateCards` – 2
  - `shuffleArray` – 3
  - `findCard` – 1
  - `shiftDownOne` – 2
  - `tripleCut` – 3
  - `countCut` – 2
- Encoder class:
  - `sanitize` – 2
  - `charToInt` – 1
  - `intToChar` – 1
  - `encodeChar` – 2
  - `decodeChar` – 2
  - `encodeString` – 2
  - `decodeString` – 2
  - `nextKeyStreamChar` – 5
  - `encrypt` – 2
  - `decrypt` – 2
- Message class:
  - `tryPosition` – 4
  - `Message` – 5

Methods were assessed independently, e.g. if `allDifferent` had a problem, that wouldn’t affect other methods that called it. A method that got all green ticks from the marking class would get full marks; otherwise a penality was applied according to the severity of the error.

There were 10 marks for code clarity. Mostly you would have lost a mark if you do something significant contrary to the CSSE Java style guide, or if the marker deemed that your code for a method was significantly more complicated than it needed to be.

The overall average was 43/55. The distribution of marks is shown below.



Sample solution

This is my solution – compare this to your own solution.

- *Deck.java*
- *Encoder.java*
- *Message.java*

The tests used for marking are contained in the following classes:

- *DeckTest.java*
- *EncoderTest.java*
- *MessageTest.java*

In addition, the tests provided as part of the project specification were also run:

- *DeckTestPublic.java*
- *EncoderTestPublic.java*
- *MessageTestPublic.java*

Experiment with the test classes, and see if you can pinpoint where your program goes wrong (if it does!). Please direct questions to help1001.

Help!

The quickest way to get help with the project is via help1001. You can ask questions, browse previous questions and answers, and discuss all sorts of topics with both staff and other students.

Please read and follow these guidelines.

- **Do not post project code on help1001.** For obvious reasons, this behaviour undermines the entire assessment process: as such you will be liable for punishment under the University's guidelines on academic conduct.
- Before you start a new topic, check first to see if your question has already been asked and answered in an earlier topic. This will be the fastest way to get an answer!
- When you do start a new topic, give it a meaningful subject. This will help all students to search and to make use of information posted on the page.
- Feel free to post answers to other students' questions. And don't be afraid to rely on other students' answers: we read everything and we will correct anything inaccurate or incomplete that may be posted from time to time.
- Be civil. Speak to others as you would want them to speak to you. Remember that when you post anonymously, this just means that your name is not displayed on the page; your identity is still recorded on our systems. Poor behaviour will not be tolerated.
- **Do not post project code on help1001.** This one is worth saying twice.

If you have a question of a personal nature, do not post to help1001: please email Lyndon or Arran instead.

Good luck!