Title: **Software Development Continuous Assessment**

Handed out date: **30th September 2025**

Submission deadline: 2nd **November 2025**

This assessment contributes **40%** of the total module mark and assesses the following **intended learning outcomes**:

- Use a software design and development method which incorporates both formal and informal techniques appropriately.
- Design and implement rigorous testing frameworks for software and be aware of the inherent limitations of tests developed.
- Deploy advanced object-oriented language concepts and techniques.
- Develop multi-program software systems.
- Follow the phases of software development.
- Recognise and evaluate different development practices, and judge their appropriateness for a specific development problem.
- Analyse and break down a problem into constituent parts.
  Compare and critically contrast different potential solutions to a problem.
- Follow the pair-programming development approach used widely in industry.

Plagiarism

This is a pair assessment, and you are reminded of the University's regulations on collaboration and plagiarism. You must avoid plagiarism, collusion, and any academic misconduct behaviours. Further details about academic honesty and plagiarism can be found at https://ele.exeter.ac.uk/course/view.php?id=1957.

Generative AI

This assessment is AI-prohibited – where the use of GenAI tools is prohibited as their use prevents achievement of the module learning outcomes. You can find further information in the university's policies around using AI in assessed work

Instructions

In this assignment you will implement an application using the various techniques that you are learning about during this module. You will tackle this task using the pair programming development paradigm that will reinforce the idea of programming as a team exercise. Please find the pair information on ELE (in the Assessment tile). Note, any changes of the members to the pairs need to be approved by the module leader, otherwise a mark of zero for the relevant teams may be applied.

1. Task Specification

You will develop, in Java, a *multi-threaded* card playing simulation. Within your design you will need to implement (at least) a thread-safe Card class and a thread-safe Player class (depending upon your design, you may also implement additional classes, for instance a CardDeck class). You will also develop an executable CardGame class.

The game has n players, each numbered 1 to n (which for clarity in the illustration below are named player1, player2, …, player$n$), with n being a positive integer, and n decks of cards, again, each numbered 1 to n (which for clarity in the illustration below are named deck1, deck2, … , deck$n$). Each player will hold a hand of 4 cards. Both these hands and the decks will be drawn from a **pack** which contains $8n$ cards. Each card has a face value (denomination) of a non-negative integer[1].

The decks and players will form a ring topology (see illustration in the figure below for the case where n = 4). At the start of the game, each player will be distributed four cards in a round-robin fashion, from the top of the pack, starting by giving one card to player1, then one card to player2, etc. After the hands have been distributed, the decks will then be filled from the remaining cards in the pack, again in a round-robin fashion.
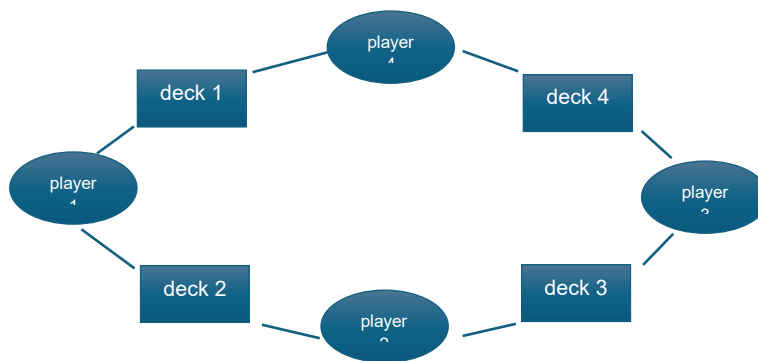

Figure 1: Topological relationship of the game players and card decks, in the situation where n = 4.

Deck definition. A **deck** is an ordered, thread-safe container (FIFO) holding card denominations from the shared 8n pack. Players **draw from the top (front)** and **discard to the bottom (back)** of decks as specified by the ring topology (player i draws from deck i and discards to deck $i+1$). Draw and discard together are treated as **one atomic action**. At game end, **each player holds exactly four cards**, and each deck's **final contents** are written to deckX_output.txt (deck sizes may vary).

---

[1] It is legal for the face value of a card exceeding n.

To win the game, a player needs four cards of the same value in their hand. If a player is given four cards which are all the same value at the start of the game, they should immediately declare this (by their thread printing "Player *i* wins", where *i* should be replaced with the player index), that player thread should then notify the other threads, and exit[2].

If the game is not won immediately, then the game progresses as follows: each player picks a card from the top of the deck to their left and discards one to the bottom of the deck to their right[3]. This process continues until the first player declares that they have four cards of the same value, at which point the game ends[4].

1.1 Game playing strategy

If a player does not start with a winning hand, they will implement a simple game strategy, as specified below (note, the strategy is *not* optimal).

Each player will prefer certain card denominations, which reflect their index value, e.g., player1 will prefer 1s, player2 will prefer 2s, etc. After drawing a card from their left, a player will discard one of their cards to the deck on their right, (e.g. player1 will draw from deck1 and discard to deck2). The card they discard must display a value which is **not** of their preferred denomination. Additionally, a player **must not** hold onto a non-preferred denomination card indefinitely, so you must implement your Player class to reflect this restriction (otherwise the game may stagnate).

1.2 Solution development

You will need to implement an executable class called CardGame, whose main method requests via the command line (terminal window) the number of players in the game (i.e. '*n*'), and on receiving this, the location of a valid input pack. The screenshot below shows how this works (note, I put the pack file in the same directory of CardGame, so there is no path needed in the example shown below.)

```
[Yuleis-Air:example_answer_CA1 yw433$ java CardGame
Please enter the number of players:
4
Please enter location of pack to load:
four.txt
```

A valid input **pack** is a plain text file, where each row contains a single non-negative integer value, and has 8*n* rows. The screenshot below shows a partial pack

---

[2] There is a chance that, two or more players are given four cards with the same value at the start of the game. You don't need to handle this situation in your development.

[3] By multi-threading, players should NOT play the game sequentially, i.e., NOT in a way that, when one player finishes actions another player starts.

[4] By multi-threading, there is a chance that, two or more players have their four cards of the same value at the same time. You don't need to handle this situation in your development.

```
14
15
8
2
2
4
13
4
13
3
14
5
3
4
9
7
5
13
9
8
14
4
12
12
6
```

After reading the input pack, the CardGame class should distribute the hands to the players, fill the decks and start the required threads for the players. If the pack is **invalid**, the program should inform the user of this, and request a valid pack file[5].

If a player does not start with a winning hand, as a player processes their hand, each of its actions should be printed to an output file which is named after that particular player (i.e. the output file for the first player should be named player1_output.txt)[6]. In game actions should be printed to the file in a similar form to the following example:

player 1 draws a 4 from deck 1

player 1 discards a 3 to deck 2

player 1 current hand is 1 1 2 4

Additionally, at the start of the game the first line of the file should give the hand dealt, e.g.

player 1 initial hand 1 1 2 3

and at the end of the game the last lines of the file should read either:

player 1 wins

player 1 exits

player 1 final hand: 1 1 1 1

if for instance player 1 wins, or

player 3 has informed player 1 that player 3 has won

player 1 exits

player 1 hand: 2 2 3 5

in the case where player 3 has won (and the values displayed match the hands and decks concerned).

There should also be a message printed to the terminal window (as is the case when a player wins immediately), i.e. if the 4th player wins, then

player 4 wins

should be printed to the screen.

---

[5] The game should not start until there are valid inputs for the 'number of players' and the 'pack' file.
[6] Each player should have its own output file, NOT a combined output file for all players.

There should only be one player declaring it has won for any single game[7]. If the game is won immediately (a winning hand is initially dealt), the output files should still be written for all players (although each file will only contain four lines). In addition to the player output files, there should also be *n* deck output file written at the end of the game (named, e.g. deck1_output.txt), which should contain a single line of text detailing the contents of the deck at the end of the game, e.g.

> *deck2 contents: 1 3 3 7*

The combination of a card draw, and a discard should be treated as a single atomic action. Therefore, at the end of the game every player should hold four cards. The program developed should follow the object-oriented paradigm.

You are encouraged to use a version control system, but this will not contribute to your final mark.

2. Submission Requirements

Your submission must include both the code part (cards.jar and cardsTest.zip) and the report in a single ZIP upload. If either component is omitted, only the submitted component will be marked and the omitted component will receive 0. You need to ZIP both parts into one file and submit it *electronically* using ELE, by 12 noon on the hand in date.

The Code Part. Your code part includes the following two files (i.e., cards.jar and cardsTest.zip).

- A copy of your finished classes in an executable *jar* file named cards.jar. The jar file should include **both** the bytecode (*.class*) and source files (*.java*) of your submission.

- A copy of your finished classes, and associated test classes and test suites, and any supporting files, in a *zip* file named cardsTest.zip. The zip file should include **both** the bytecode (*.class*) and source files (*.java*) of your submission (plus any testing files the tests may rely upon), and a README file, detailing how to run your test suite.

The Report Part. The report, with minimum 2 cm margins and 11-point text, should contain the items listed below.

- A (max 2 page) document *detailing* your design choice and reasons with respect to both your production code, and any known performance issues. This part of the document should be no more than two sides of A4.

- A (max 3 page) document *detailing* the design choice and reasons with respect to your tests. You may use either of the JUnit 4.x or 5.x frameworks, but you should explicitly detail which framework you are using in your document. This part of the document should be no more than three sides of A4.

---

[7] By multi-threading, there is a chance that, two or more players declare they have won the game simultaneously. You don't need to handle this situation in your development. You can simply give it another run.

- A (max 1 page) development **log**, which includes date, time and duration of pair programming sessions, and which role(s) developers took in these sessions, with each log entry signed by both members (using your candidate number as your signature). This part of the document should be no more than one side of A4.

3. Some Questions You May Have

1. Which card should a player choose to discard?

Answer: You can randomly choose one which is not the preferred denomination to discard.

2. Can a player who is given four cards which are all the same value at the start of the game, but the value is not their preferred denomination, still win?

Answer: Yes, it still can win the game.

3. How many output files from the production code?

Answer: 2*n*, where *n* is the number of players.

4. Do I need to use Javadoc?

Answer: No, you don't need to.

5. My code reads in a valid pack file which doesn't contain the cards that allow a winning hand. What should I do?

Answer: It's normal. You don't need to do anything. The code and pack are two separate things. You need to make sure your code is correct, which means if it takes in a valid pack that allows a winning hand, it can output a winning hand.

6. Do I need to test private methods?

Answer: Yes, you can consider using Java Reflection.

Your CA will be marked based on the criteria in Section 4.

Marking criteria

This assessment will be marked using the following criteria.

Marking Scheme	Description	Mark
The Report		
If the report is missing: We will award **0** for the report criterion (**40%**) and mark the code criteria normally (up to **60%**).		
Structure and contents of the report	The report is well structured and presented. The design is well explained, matches the specification provided and the implemented code. The development log contains the specified information.	40%
The Code		
If the code part is missing (either cards.jar or cardsTest.zip not present): We will award **0** for all code-related criteria (Implementation & Testing 35%, Exception handling 10%, Code comments 10%, README 5% = **60% total**). Only the report (up to 40%) may be marked.		
Code comments	Code comments are useful and informative, and at the appropriate level (i.e., it should not contain spurious comments, or ones that do not serve an explanatory purpose).	10%
Production Code. Implementation & Testing.	The code is well structured and presented, with a coherent design and clear management of object states. The program is thread-safe, produces output and takes input in the formats specified. The JUnit tests are well formulated and cover the code well.	35%
README file.	The file details all that is required to run your test suite.	5%
Production Code. Handling exception states.	The code deals smoothly with exceptional inputs and is robust in use.	10%
Penalties		
Penalty	Submission is greater than the stated number of pages (6), excluding the ELE sheets	No penalty applied, but only the first 6 pages will be marked.

Further Information and Guidance for students on submitting group work:

Only one student per pair should submit the completed file. Nominate someone in your pair to do this and make sure you arrange a time, before the deadline, to confirm that this has been done. If possible, do this together (or via a shared screen in Teams) so you can

all check that the submission is correct, it has been uploaded to the correct link, and that the file hasn't been corrupted.

You MUST declare yours and your pair programming partner's details.