SPL - ASSIGNMENT 2

Java Concurrency, Synchronization and Testing

Lecturer in charge: Marina
TAs in charge: Tal Raveh, Pan Eyal

Publication date: 27.11.2022

Submission deadline: 18.12.2022 23:59

Document version: 1.0 - 27.11.2022

Before You Start

(We know you want to skip this part, but please read it carefully!)

The goal of the assignment is to practice concurrent programming on Java 8 environment, and give you basic experience with unit testing. This assignment requires a good understanding of Java Threads and Java Synchronization. Make sure you revise the lectures and practical sessions that cover these topics.

You are free to develop your project in an environment of your choice, but the projects will be tested and graded on a CS LAB UNIX machine. It is therefore recommended that you run your assignment on a lab unix machine before submission.

The Q&A of this assignment will take place at the course forum only. Critical updates about the assignment will be published in the assignment page on the course website. These updates are mandatory. It is your responsibility to be updated.

Guidelines for using the forum

Read previous Q&A carefully before asking a new question! Repeated questions might be left unanswered.

You are **not** allowed to post any kind of solution and/or source code in the forum as a hint for other students. In case you feel that you have to discuss any such matters, please use the staff reception hours.

Yair is the only staff member who can approve extensions. In an appeal for an extension, please contact him directly.

Good Luck!

1) GENERAL GUIDELINES

- Read the javadocs of all the interfaces we provide you with.
- Please stick to the java documentation of the classes and methods.
- **You are allowed to make any change you want to/in the following files/directories:**
  - *pom.xml*
  - *src/main/java/bguspl/set/ex*
  - *src/test*
- These changes may include (but not limited to):
  - Using any Java synchronization mechanism you want.
  - Throwing and handling exceptions.
  - Changing the classes or writing new ones in the aforementioned directories.

  To clarify - this code (pom.xml, Dealer.java, Player.java, Table.java and the unit test files) is there for your convenience, to assist you in writing the solution. You are not obligated to use it if you do not wish to.
- *Make your code simple and straightforward.*
- *You must use synchronization only where it is needed* - excessive and redundant synchronization may affect your grade.
- *Your code should be efficient* - inefficient code may affect your grade. **However, efficiency must not come at the cost of correctness.**

2) INTRODUCTION

In the following assignment you are required to implement a simple version of the game "Set". A description of the original game can be found here: Set Card Game. For reference only, you can also watch the next video for a more intuitive understanding of the game: How to play Set (keep in mind though, that we use slightly different rules in our implementation).

All of the UI (User Interface), graphics, keyboard handling, etc. were already written for you - you only need to implement the game logic parts.

3) OUR VERSION OF THE SET GAME

The game contains a deck of 81 cards. Each card contains a drawing with **four** features (color, number, shape, shading).

The game starts with 12 drawn cards from the deck that are placed on a 3x4 grid on the table. The goal of each player is to find a combination of three cards from the cards on the table that are said to make up a "legal set".

A "legal set" is defined as a set of 3 cards, that for each one of the four features — color, number, shape, and shading — the three cards must display that feature as either: **(a) all the same**, or: **(b) all different** (in other words, for each feature the three cards must avoid having two cards showing one version of the feature and the remaining card showing a different version).

The possible values of the features are:

- The color: red, green or purple.
- The number of shapes: 1, 2 or 3.
- The geometry of the shapes: squiggle, diamond or oval.
- The shading of the shapes: solid, partial or empty.

For example:

Example1: these 3 cards **do form** a set, because the shadings of the three cards are all the same, while the numbers, the colors, and the shapes are all different.

Example 2: these 3 cards **do not form** a set, because although the numbers of the three cards are all the same, and the colors, and shapes are all different, only two cards contain the same shading, and the third do not.

The game's active components contain the “Dealer” and the “Players”.

The players play together simultaneously on the table, trying to find a legal set of 3 cards. They do so by placing tokens on the cards, and once they place the third token, they should ask the dealer to check if the set is legal.

If the set is not legal, the player gets a penalty, freezing his ability of removing or placing his tokens for a specified time period.

If the set is a legal set, the dealer will discard the cards that form the set from the table, replace them with 3 new cards from the deck and give the successful player one point. In this case the player also gets frozen although for a shorter time period.

To keep the game more interesting and dynamic, and in case no legal sets are currently available on the table, once every minute the dealer collects all the cards from the table, reshuffles the deck and draws them anew.

The game will continue as long as there is a legal set to be found in the remaining cards (that are either on table or in the deck). When there is no legal set left, the game will end and the player with the most points will be declared as the winner!

Each player controls 12 unique keys on the keyboard as follows. The default keys are:

Player A				Player B			
Q	W	E	R	U	I	O	P
A	S	D	F	J	K	L	;
Z	X	C	V	M	,	.	/

The keys layout is the same as the table's cards slots (3x4), and each key press dispatches the respective player's action, which is either to place/remove a **token** from the card in that slot - if a card is not present/present there.

The game supports 2 player types: human and non-human.

The input from the human players is taken from the physical keyboard as an input.

The non-human players are simulated by threads that continually produce random key presses.

4) THE GAME DESIGN

4.1) The Cards & Features

Cards are represented in the game by int values from 0 to 80.

Each card has 4 features with size 3. Therefore, the features of a card can be represented as an array of integers of length 4. Each cell in the array represents a feature. The value in the cell represents one of the 3 possible values of the feature. In that way each card can get a unique id based on the features that described it and vice versa. The id of each card is calculated as follows:

$3^0 * f_1 + 3^1 * f_2 + 3^2 * f_3 + 3^3 * f_4$ where f_n is the value of feature n.

4.2) The Table

The table is the data structure for the game. It is a passive object that is used by the dealer and the players to share data (more on the dealer and players later).

The table holds the placed cards on a grid of 3x4, and keeps track of which token was placed by whom.

4.3) The Players

For each player, a separate thread is created to represent the player entity.

The player object manages the player data (such as the player id and type - human/non-human).

For non-human players, **another thread** is created to simulate key presses.

You must maintain a queue of incoming actions (which are dispatched by key presses – either from the keyboard or from the simulator). The queue size must be equal to the number of cards that form a legal set (3).

The player thread consumes the actions from the queue, placing or removing a token in the corresponding slot in the grid on the table.

Once the player places his third token on the table, he must notify the dealer and wait until the dealer checks if it is a legal set or not. The dealer then gives him either a point or a penalty accordingly.

The penalty for marking an illegal set is getting frozen for a few seconds (i.e. not being able to perform any actions). However, even when marking a legal set, the player gets frozen for a second.

4.4) The Dealer

The dealer is represented by a single thread, which is the main thread in charge of the game flow. It handles the following tasks:

- Creates and run the player threads.
- Dealing the cards to the table.
- Shuffling the cards.
- Collecting the cards back from the table when needed.
- Checking if the tokens that were placed by the player form a legal set.
- Keeping track of the countdown timer.
- Awarding the player with points and/or penalizing them.
- Checking if any legal sets can be formed from the remaining cards in the deck.
- Announcing the winner(s).

Notes:

1. When dealing cards to the table and collecting cards from the table the dealer thread should sleep for a short period (as is already written for you in Table::placeCard and Table::removeCard methods provided in the skeleton files).
2. Checking user sets should be done “fairly” – if 2 players try to claim a set at roughly the same time, they should be serviced by the dealer in “first come first served” (FIFO) order. Any kind of synchronization mechanism used for this specific part of the program must take this into consideration.

4.5) The Graphic User Interface & Keyboard Input

Note: do not make any changes to any of the components in this section.

4.5.1) The User Interface

All the Graphic User Interface functionality was written for you and available via the *UserInterface* interface. You only need to invoke this functionality when appropriate.

The UI class opens a new window on screen with the following components:

As mentioned above, you must explicitly call the corresponding method to update the graphics that are displayed to the user every time a change is required. Review the interface methods in *UserInterface* and make sure you understand them.

The implementation of the *UserInterface* interface is in the *UserInterfaceImpl* class. They have been written in a generalized manner, based on the fields of the Config class that we provided you with.

The configuration settings of the game reside in the file *config.properties* and is loaded into the Config class.

4.5.2) The Input Manager

The handling of the input from the keyboard has also been written for you in the *InputManager* class. It will automatically call Player::keyPressed method and pass as an argument the **slot number** of the card that corresponds to the key that was pressed.

The slot number is: $\boldsymbol{column + total\ columns * row}$

4.5.3) The Window Manager

When the user clicks the close window button, the class *WindowManager* that we provided you with, automatically calls Dealer::terminate method of the dealer Thread, and Player::terminate method for each opened player thread.

4.6) Other Components Supplied for You

Note: do not make any changes to any of the components in this section.

4.6.1) Main class

Loads all the required components and creates and runs the dealer thread.

4.6.2) Env class

Helper class for sharing the Config, UserInterface and Util classes.

4.6.3) Config class

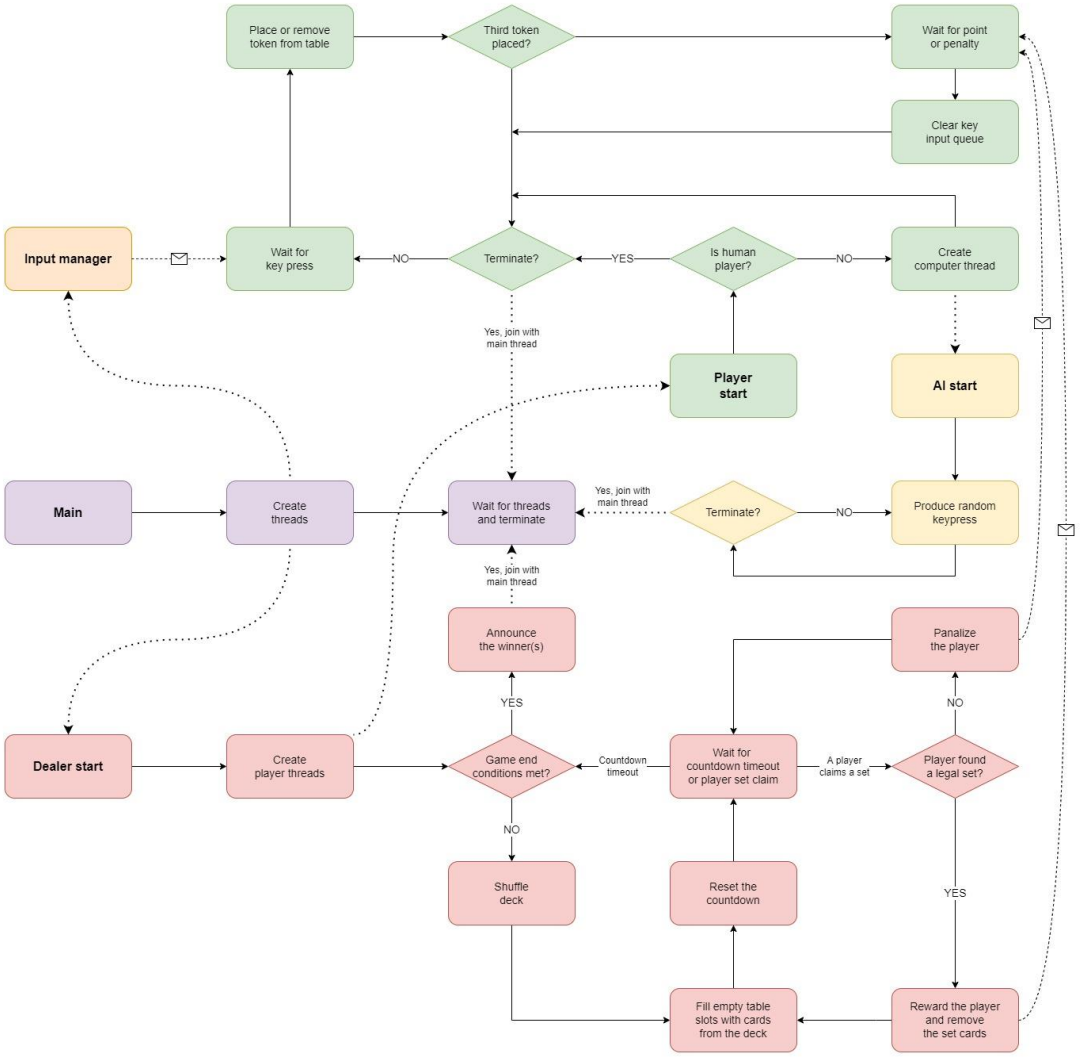
Loads and parses the configuration file. The configuration object is passed to any class that may need it. You should use the configuration fields where applicable. A bonus of +3 points will be awarded for fully supporting all configuration fields, and avoiding use of any magic numbers.

4.6.4) Util class

This class contains several utility methods that you may use freely in your implementation (converting card id to array of features and finding/testing if a collection of cards contains/is a legal set).

5) PROGRAM EXECUTION / FLOW

Following is a diagram of the general flow of the program. Following this flow should provide a fluent game experience and should also produce the correct event log output. Note that this diagram only describes the general flow and does not address how to handle synchronization, graphic display updates and such.



Another visual demonstration of the game can be found in a helper video we uploaded to the moodle website.

6) BUILDING WITH MAVEN

In this assignment you will use Maven as your build tool. You should read about how to use it properly (we recommend Apache Maven Tutorial | Baeldung). We have supplied you with a *pom.xml* file that you can use to get started. You should learn about this file and how to add dependencies to it if needed.

7) UNIT TESTING

Testing is an important part of software development. It helps to ensure the project behaves as expected. In this assignment, we use JUnit for testing.

7.1) JUnit tests

You are required to write at least 2 new unit test methods for a method or methods of your choice for each of the classes:

- Table
- Dealer
- Player

You are not required to write more than these 2 unit tests, but you may find that writing more tests will most likely **save you time** overall – despite the time it takes to write them.

7.2) Mockito

Mockito is a *mocking framework* that you may use your unit test methods *if you wish*.

Mocking helps test class functionality in isolation. Mock objects are objects that are created in test environment to replace actual objects. They implement the same interface as the actual objects, but either do nothing, or behave in a predefined manner which is tailored to the specific needs of the test case.

Mock objects are created in runtime only during tests and never affect the execution of the program otherwise (in fact, they should not even be accessible at all during the normal execution of the program).

You can read more about the Mockito mocking framework here:

- Getting Started with Mockito | Baeldung
- Mockito and JUnit 5 - Using ExtendWith | Baeldung

To help you understand the expected tests format, we have provided you with an example of unit tests with JUnit (without *Mockito)* in TableTest.java, and another example (with *Mockito*) in PlayerTest.java

8) SUBMISSION INSTRUCTIONS

8.1) The Directory Structure

Attached to this assignment is a (partial) Maven project you can use to start working on your assignment. The project's directory structure is the most commonly used for Maven projects - you may not change it.

Included in the attached project are:

- Interfaces and skeleton of the entire project.
- The directory structure you must use for the submission.
- A Maven pom.xml file.

8.2) The Submission Process

- Submission is done only in pairs. If you do not have a pair, find one. You need explicit authorization from the course staff in order to submit without a pair. You cannot submit in a group larger than two.
- Submit a *zip* file with all your code.
  The file should be named "s*tudentID1_studentID2.zip*".
  Note: other types of archive files, such as '*.rar*', '*.bz*', or anything else which is not a '*.zip*' file will not be accepted.
- The contents of the zip file should be as follows:
  - The *pom.xml* file used to build the project.
  - The *src* directory of your project.
- **However, only the contents of the following directories will be tested:**
  - *src/main/java/bguspl/set/ex*
  - *src/test*

  All files in other directories will be overwritten with the original skeleton files!

  Obviously, changing the signatures of the constructors of Table, Dealer and/or Table used in Main::main will break compilation once the files in src/main/java/bguspl/set are overwritten. Make sure you do not change these signatures.
- The project will be rebuilt using the Maven *pom.xml* file.
- Extension requests are to be sent to Yair. Your request email must include:
  - Your name and your partner's name.
  - Your id and your partner's id.
  - Explanation of the reason for the extension request.
  - Official certification.
- **Requests without a compelling reason will not be accepted.**

9) GRADING

The assignments will be checked and graded on Computer Science Department Lab Computers so be sure to test your program thoroughly on them before your final submission.

Grading will incorporate the following:

- The application design and implementation.
- Tests will be run on your application.
- Running the game with 4 non-human players and a configuration of 0 penalty/point freeze time (and must end successfully and in a reasonable time).
- Graceful termination of all threads.
- Liveness and deadlocks, the synchronization mechanisms you chose to use and where you have used them and the reasons behind your decisions.
- Points will be reduced for redundant and/or too broad synchronization.
- A bonus of +3 points will be awarded for fully supporting all configuration fields, and avoiding use of magic numbers.
- A bonus of +2 points will be awarded for terminate the threads **gracefully** and **in reverse order** to the order they were created in.

WE WISH YOU GOOD LUCK

AND HOPE YOU ENJOY DOING THIS ASSIGNMENT