General computer science 1
Summer semester 2023

Programming Project Version: June 5, 2023

TECHNISCHE UNIVERSITÄT DARMSTADT

Processing time: 06/07/2023 to 07/05/2023

organizational

Important: First read the following formalities carefully. If these are not observed exactly, we may not be able to evaluate your submission. Read through the task statement in its entirety before beginning implementation.

- Note all the information on the programming project on the page: **https://moodle.informatik.tu-darmstadt.de/mod/assign/view.php?id=55356**
- Use our **guidelines** and enter your name and your matriculation in the included **README.txt** file number and your TU-ID (as well as the data of your partner in groups of two!).
- In order to submit the ProPro, you must be registered in a ProPro group in Moodle. The submission in Moodle is always for the entire group. In groups of two, it is sufficient if only one person uploads the solution.
- Submission of the project (no later than **07/05/2023** at **10:00 p.m.)**
  - Pack your entire project directory in a standard ZIP file. The name of the file must consist of the initials of the group members and the development environment used, e.g. B. **MK_bluej.zip** or **MK_eclipse.zip.** In the case of groups of two, both names accordingly, e.g. B. **MK_CK_bluej.zip.**
  - Upload the ZIP file to our Moodle platform. **No** ZipX, 7z or RAR archives! More detailed instructions on how to create the submission can be found on Moodle in the programming project area.
  - In any case, try the delivery well before the deadline! You can multiple times hand over. If there are multiple submissions, only the last one will be counted.
- A total of 25 points can be achieved. Defects in formatting or comments lead to deductions (see 6). Submissions that cannot be compiled receive **0 points.** The project should ideally be edited with **BlueJ** or **Eclipse .** It is allowed to use another IDE as long as the project can run on **BlueJ** or **Eclipse** and the README.txt file is included accordingly.
- **Caution:** The Department of Computer Science attaches great importance to compliance with scientific ethics, including the strict prohibition of plagiarism. By submitting your solution, you confirm that you are the sole author of the entire material. This means that "copying" (also alienated), "having it copied" and "comparing solutions", whether by passing on program code or verbally, are prohibited. Make sure not to leave your notebook, computer or USB stick unattended. For more information on this topic, see:

  https://www.informatik.tu-darmstadt.de/studium_fb20/im_studium/studienbuero/ plagiarismus/index.de.jsp

  - Consequences: **0 points** in the programming project or even in the entire exam!
  - Of course, in addition to a manual check, we use a very reliable plagiarism checker Software.

Important instructions

1) The project is not fully compilable at the beginning, this is intentional! The classes are still missing and methods which you should write yourself in 1, **2** and **3 .**

2) You can edit the project in the usual **BlueJ** development environment . You are also welcome to take the opportunity to get to know **Eclipse** (a more powerful IDE) and edit the project in it.
You can read more about the differences between **BlueJ** and **Eclipse** and the benefits of **Eclipse** on the Moodle forum.

3) The project requires at least **Java version 11.** A pre-installed Java version is no longer required for **Eclipse** or **BlueJ .** Usually you can just use the latest Java versions of **Eclipse** or **BlueJ .** If there are problems, you can also install and use a standalone Java version. Check your installed Java version in the following way:

- **Windows1 :** *Start* ÿ *All Programs* ÿ *Accessories* ÿ *Command Prompt* and type:

  java -version

- **Linux & Mac:** Open a terminal and type *java -version* and press Enter.

If you have an outdated Java version, you must first update to Java 11 (or newer). You will find download files and installation instructions for this, e.g. B. **at https://www.oracle.com/de/java/technologies/downloads/**

Note for Mac users with Apple Silicon (e.g. M1): Although there are now adapted Java versions for Apple Silicon, an Intel variant must still be used for this project in **Eclipse .** The integration is a bit more complicated, so there is a tutorial on Moodle in the **ProPro setup section.**

Note for Linux users: In our experience, there may be problems running this project on **Linux .** Make sure you are using the latest versions of **Eclipse** or **BlueJ .** You should also not use a pre-installed Java version, but the current Java version of your IDE. If the programming project should not work like this, we have created a separate **Linux guide** for you , which you can use to install and work with a specific Java version. The **Linux instructions** can be found in the Moodle forum in the programming project section.

4) To correctly integrate the project into the development environment, please proceed as follows:

- **Eclipse:** First, download the **propro_eclipse.zip** file from our moodle platform . public name **Eclipse** and click:

  File ÿ *Import* ÿ *General* ÿ *Existing Projects into Workspace* ÿ *Select archive file* ÿ *Browse*

  Select the **propro_eclipse.zip** file you just downloaded and then click *Finish.*

- **BlueJ:** First, download the **propro_bluej.zip** file from our moodle platform . Unzip the archive and start BlueJ. You now have to add the required libraries.
  To do this, click on:

  Tools ÿ *Preferences…* ÿ *Libraries* ÿ *Add File*

  The three required files **eea.jar, slick.jar** and lwjgl.jar are located in the subfolder **libs** in the unpacked project folder . Add these! After that, BlueJ needs to be restarted. Once you have done this, you can now select the unpacked project folder with *Project* ÿ *Open Project…* and add the project in this way.

1If you're using Windows 10 or later, just type *command prompt* in the search bar .

Requirements

After you have observed the important information, you should now be able to open the programming project. It contains all required classes in subpackages of the **tud.ai1.shisen package.** During the programming project, you will only work in the **model** and **util** subpackages (you can ignore the other packages). Most of the classes and methods are already complete. Unless explicitly stated otherwise, all methods and classes you implement should be **public** and all class and instance variables should be **private .**

Also note that all parameters passed to a method must be checked to ensure that they can be used, even if this is not explicitly stated in the task! This means in particular that you should prevent possible **NullPointerExceptions !** Some methods are not yet implemented or only contain a "dummy implementation". It is your job to complete these methods (and pay attention to comments and formatting).

All other methods and classes of the default must remain untouched, also no additional data fields of any visibility are allowed!

Introduction - The game *Shisen*

Shisen is a heavily modified version of the classic Chinese board game Mahjongg. The game includes 36 different squares, each of which is placed four times on the playing field. The playing field thus consists of a total of 144 fields, arranged in an 8x18 matrix. The aim of the game is to remove all squares from the playing field, whereby two identical squares can always be removed at the same time. Two fields can only be taken if they can be connected with a maximum of three lines. The lines must be either horizontal or vertical (not diagonal) to the playing field and must not cross any other fields. In particular, this means that adjacent squares can be removed directly, regardless of where they are on the playing field. Lines may exceed the edge of the field by one field.

Note: Don't be surprised if a game cannot be completely solved. Due to the fact that there are two pairs of each field, it is possible that the game cannot be completely solved if a pair is solved "wrongly".

You can find more information about the game and a variant here2 :

Explanations: **https://dkmgames.com/shisen/**
Example: **https://dkmgames.com/shisen/shisenplay.htm**

Note that the exact specifications must be taken from the task.

In the main menu (Figure 1a) you can start a new game or view the *high scores .* If you want to start a new game, just click on the *Start Game button.* On the playing field itself (Figure 1b) you can turn the cards over with the left mouse button or activate a cheat with the *Hint, Solve Pair* or *Find Partner* buttons , which gives hints or, for example, reveals a matching pair. If you activate a field and then click on a suitable partner, both fields will disappear (Figure 1c). In addition, you can see the time that has already elapsed and the number of points you have achieved (score) in the upper right-hand corner. The top ten entries can be found in the high score overview (Figure 1d).

Note: To start the game, compile all the source code files in the subpackages and start the **main** method in the **Launch** class in the **view** subpackage .

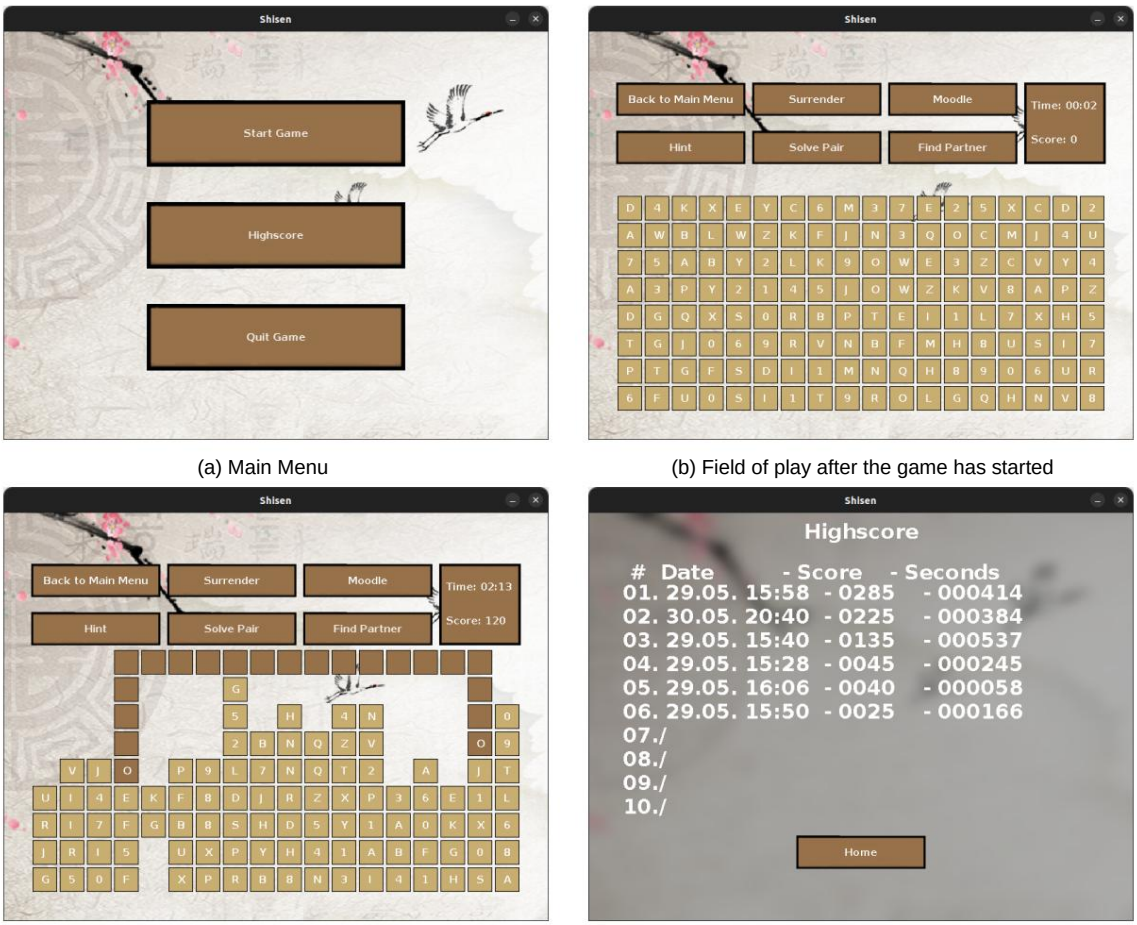
1 enumeration type (0.25 points)

As already mentioned, the entire project is not yet compileable. Therefore, some errors are also displayed in the source code. One of the reasons for this is that the **TokenState** enumeration type does not yet exist. In this task you should implement this type.
The **TokenState** enumeration type represents the state of a field and is used to mark it.
A field can have four states: *Initial State, Clicked, False ,* and *Solved.*

a) In the **model** package, create an enumeration type (= **enum)** named **TokenState.** This should be the following possess values:

DEFAULT, CLICKED, WRONG, und **SOLVED.**

2However, please note that the linked pages describe a specific description of the game for an accurate implementation. Our Project does not necessarily follow this exact description!



(a) Main Menu

(b) Field of play after the game has started

(c) Playing Field During Play

(d) Highscores

Figure 1: Overview of the individual game states.

2 Token (0.25 + 0.25 + 0.25 + 0.5 + 0.5 + 0.25 = 2 points)

In this task you should now create the **Token** class , which stands for a single field on the entire playing field.

a) In the **model** package, create the **Token** class that implements the Interface3 **IToken** and add the following five private instance variables to it:

- **counter:** This variable should be of type **int** and static. It is used to generate unique IDs, wes because it must also be initialized with **0** when it is declared .
- **id:** An **int** variable that should also be immutable. This variable enables the individual field to be clearly identified on the entire playing field.
- **state:** An array of type **TokenState.** It describes the current state of the field.
- **value:** This variable should be of type **int** and unchangeable. It describes the value of a field.
- **pos:** A **Vector2f** variable that in-game specifies the location of each square on the board.

3At this point, the programming project anticipates something from the lecture on inheritance. An interface ensures that classes provide predefined methods. In order for a class to inherit from an interface, the keyword **implements** and then the name of the interface must follow the name of the class in the class header. Every method not implemented in the interface but declared must be implemented in the inheriting class.

Note: The static type of the **pos** variable should be **org.newdawn.slick.geom.Vector2f .** (However, the variable **pos** should **not** be static!) Pay attention to the necessary import statements!

b) Create a constructor of the **Token class.** This should only have three parameters, which are of type **int, TokenState** and **Vector2f .** With these three parameters you initialize the instance variables **value, state** and **pos.** Note that in this constructor you must also set the value of the array **id** to the value of **counter** and then increment **counter** by **1 .**

c) In order not to always have to specify all values when creating an object of this class, another constructor should be implemented in addition to the constructor created in 2b, which only expects an **int** parameter. This parameter specifies the value for the instance variable **value .** Furthermore, the data field **state** must be initialized with **TokenState.DEFAULT** and **pos** with a new object of the **Vector2f** type with the parameters **0, 0 .** The data fields **id** and **counter** must be initialized exactly as in 2b.

Note: Other constructors of the same class can also be called from within a constructor.

d) Add the four getter methods **getValue(), getTokenState(), getID()** and **getPos()** to the class added. These should return the corresponding instance variable.

Note: If a method that was implemented in the class is already declared in the interface, it must be marked with the annotation **@Override** in the class so that it can be overwritten.

e) Write the getter method **getDisplayValue(),** which gets the *DisplayValue* of the **TokenDisplayValueProviders** returns.

Note: The **TokenDisplayValueProvider** is implemented as a singleton design pattern. This means that no objects of the class can be created, instead the instance of the object must be fetched with the **TokenDisplayValueProvider.getInstance()** method each time it is used in order to display Opera to be able to carry out operations.

Note: This method is also already declared in the **IToken** interface . Accordingly, an annotation must also be used here.

Note: Pay attention to the necessary import instructions!

f) Write the two setter methods **setTokenState(TokenState newState)** and **setPos(Vector2f newPos).** The setters set the appropriate instance variable in the **Token** class to the value passed to them.

3 Grid (0.5 + 1 + 0.5 + 0.5 + 0.25 + 0.5 + 0.5 + 4 = 7.75 points)

In the **model** subpackage you will find a **Grid** class that implements the **IGrid** interface . Some of the required methods in the class are already fully implemented, the remaining methods are only available as dummy implementations and should be implemented in this task.

The **grid** is the representation of the playing field on which the fields (tokens) are placed. (or rather that consists of the fields). It stores the relationship between the fields, the tokens selected, and the highs to date score.

You can think of the **grid** as a kind of matrix containing 8x18 tokens that represent different game pieces. Overall, however, the **grid** has a size of 10x20 tokens. There are border tokens around the normal tokens you click while playing, which have specific properties and form the border of the playing field. These edge tokens are in the solved state, which means they cannot be seen. They serve to create a starting path around the playing field through which the normal tokens can be solved.

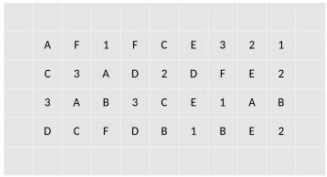
The tokens have a value and a display value. The display value is either a number from 0-9 or a letter from AZ and is assigned to the tokens after they have been successfully read from a map file. The map file stores the values of the tokens that differ from the display values (more on this in task 3h)). The edge tokens have a value of -1 and are not included in the map file.

Figure 2 serves as an example to illustrate the structure of the **grid .** It shows a smaller **grid** with its

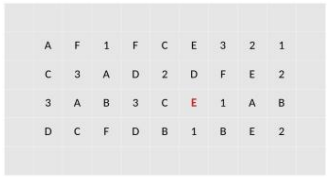
individual fields. Note, however, that the actual in-game **grid** has a different size and values.
Figure 2d illustrates how individual fields in the **grid** can be accessed. Access is via the notation **grid[x][y].** For example, if you want to access the token highlighted in red, this is equivalent to **grid[6][3].** Note that the actual in-game coordinates depend on the size and values of the **grid .**

(a) **Grid:** Values of the tokens without border = 4x9 matrix

(b) **Grid:** Values of the tokens with Rand = 6x11 Matrix



(c) **Grid:** display values of the tokens



(d) **Grid:** access token = grid[6][3]

Figure 2: Exemplary structure of a 4x9 (or 6x11) **grid.**

a) In addition to the existing instance variables, declare the following three private variables in of the Grid class .

- **grid:** This variable should be of type **IToken[][]** and static. The 2D array is intended to hold the information about the playing field itself.

  Note: Fields at the edge of the playing field should be initialized with the value **-1 .**

- **selectedTokenOne.** This variable should be of type **IToken** and represents the first selected Token.

- **selectedTokenTwo.** This variable should be of type **IToken** representing the second selected Token.

Note: Both **selectedToken variables** should be initialized with **zero** when they are declared .

b) Implement a constructor for the **Grid class,** which receives the file path to a map file as **a string .** The method creates the grid using the **parseMap(String path)** method (see 3h)), calls the pre-implemented method **fillTokenPositions() ,** which is required for determining the path, and initializes the variable **score** to a meaningful initial value.

Note: When initializing the variable **score,** consider where in the project an initial value could already be specified.

Note: You can use the **parseMap(String path)** method in this task, although it will be implemented later.

c) Implement the public method **IToken getTokenAt(int x, int y),** which returns the token at the given coordinate in the grid. It is important to note that invalid values must be intercepted. If an invalid value is passed, they return **null .**

d) Implement the public getter methods **getGrid()** and **getActiveTokens()** for the respective Data fields **grid** and **activeTokens.**

Note: In the getter for the **activeTokens ,** please return the content of the two data fields **selectedTokenOne** and **selectedTokenTwo** as an IToken array.

e) Implement the public method **boolean bothClicked(),** which returns whether two tokens have been selected (selected).

Note: You can do this simply by checking whether the two fields **selectedTokenOne** and **selectedTokenTwo** are set. There is no need to check the status of the **tokens !**

f) Implement the public method **deselectToken(IToken token)** which, given the given token is selected, sets its state **(TokenState)** to **DEFAULT** and deselects it.
Also implement the public method **deselectTokens(),** both of which selected before
Token deselected.

Note: Both methods should not have a return value.

g) Implement the public static method **boolean isSolved(),** which only returns **true** if all squares on the playing field are solved (i.e. have the **TokenState TokenState.SOLVED).**

h) Study the format of a map file *(/assets/maps/001.map).*
Implement a private method **Token[][] parseMap(String path)** that gets the path to a map file and uses it to construct a 2-dimensional array of **tokens** to represent the playing field. All normal tokens (i.e. no border tokens, see note) have a value , which is in the map file, a state **(TokenState),** which is initially **DEFAULT** for all normal tokens , and a position on the playing field as **Vector2f.** The first dimension should represent x and the second y.
Note that you can use the **IOOperations** helper class to get the contents of files as a string, and that individual lines in map files are separated with the **System.lineSeparator()** string (even if you don't see the corresponding symbols in the text editor).

Note: Note that the edge of the playing field consists of solved **(TokenState.SOLVED)** tokens, which are not included in the map files! So you don't need an 8x18 matrix, but a 10x20 matrix, which consists of a border on the outer fields. This edge is characterized by the fact that the outer fields have the value **-1 .**

Note: The **String.split(String)** method might be useful for this task.

Note: We do not want this method to be accessible from outside of this class.

4 High score (2.5 + 2 = 4.5 points)

In the **model** package you will find the **HighscoreEntry** and **Highscore classes.** These stand for an individual entry in the high score list as well as for the entire high score. Once you start implementing these classes, you might run into problems with the high score file as it might have been written incorrectly. If new errors or exceptions occur there, you could first try deleting the *assets/highscore/highscores.hs* file. The file will be recreated when a new high score is achieved.

4.1 High Score Entry (0.5 + 0.25 + 0.75 + 0.5 + 0.5 = 2.5 points)

First, look at the **HighscoreEntry** class . First, familiarize yourself with the class, looking in particular at the instance variables so that you understand their meaning. The class has three private instance variables that you should care about.

- **LocalDateTime date:** This field represents the date and time of the game being played.
- **double duration:** This data field represents the elapsed time in seconds.
- **int score:** This entry represents the score achieved.

If you look at Figure 1d, a **HighscoreEntry** would represent one of the entries, e.g. B. for the first entry with date and time 29.05. 15:48, the achieved score of 285 and the required time 414 (displayed in seconds!).

The methods are already specified in the class with a "dummy function". Now implement these methods according to the following:

a) Implement the **validate method.** This should ensure the following for the parameters passed:

- The date cannot be **null .**
- The duration must not be negative.
- The score must not be negative and must not be greater than 1000.

If one of the conditions is violated, the method should throw an **IllegalArgumentException .**

Note: Keep in mind that a **LocalDateTime** can also be **null** and you must catch this.

b) Implement the constructor **public HighscoreEntry(LocalDateTime date, int score, double, duration).** This should first check whether the parameters passed are valid. If this is the case, the instance variables should be initialized with the appropriate parameters.

c) Implement the second constructor **public HighscoreEntry(String data).** This receives the high score entry as a **string** in the format "dd.MM.yyyy HH:mm;score;duration". The individual entries are separated by a ";" separated. The constructor should check whether the parameter is valid - i.e. conforms to the stated format and is neither **null** nor empty - and also to ensure that all entries are present. If this is not the case, an **IllegalArgumentException** should be thrown.

If all entries are present, they should be validated, as in the first constructor. Finally, the data fields should be initialized according to the associated values.

- Valid parameters: "01/04/1996 07:07;123;851256.0", "2018/07/23 14:23;20;49546.0", "23.05.2019 18:20;42;142569.0".
- Invalid parameters: "", "iD-12", ";7;", " ;7;15;1300", "5.1.;6;-1", etc.

Hint: You can use the **split()** method of the **String** class to split the **string !** Another possibility would be to implement it with a loop. You also need the **parse()** method of the LocalDateTime class .

d) Write the **public boolean equals(Object obj) method.** This compares the current one **HighscoreEntry** with the passed parameter. It should therefore first be checked that the parameter is not **null** and is of the **HighscoreEntry** type . If this is the case, the values of the data fields should be compared. The method should only return **true** if all data fields are the same . In all other cases, the return is **false.**

e) Implement the **compareTo(HighscoreEntry o) method.** The method is used by the game to sort the high score entries correctly. The method compares the values of the high score entries and has the following returns:

1. If the score of the current **HighscoreEntry** is less than the score of the passed **HighscoreEntry,** the return should be any positive number.
2. If the score of the current **HighscoreEntry** is greater than the score of the passed **HighscoreEntry,** the return should be any negative number.
3. If the score is the same, the duration is compared inversely to 1. and 2. This means that a shorter duration should be considered dominant. But if the duration is the same in the end, the return should be 0.

4.2 High score (1 + 1 points)

Finally, two methods have to be implemented in the Highscore class for the **high** score. These are responsible for first adding elements and then saving them in the *highscores.hs* file.

a) Implement the **addEntry(HighscoreEntry entry) method,** which adds the new entry to the appropriate highscore list if it still fits in the list. If there is no more space in the list (ten entries already exist) and the new high score is at most as good as the last existing entry, then the new entry should not be added to the high score list. Otherwise, the entry should be added and the high score entries sorted so that a sorting as shown in figure 1d comes about. The list should therefore be sorted first by score and only if the score is the same by the duration of the respective game. In the end, however, the list should again consist of a maximum of the ten best entries.

Hint: First get the matching list and see how many items are already in the list.

Note: The **Collections** class has methods that can be useful in completing this task.

b) Finally, the **saveToFile(String fileName)** method should be implemented. This is responsible for constantly writing all high score entries (from the list) to the high score file. To do this, all entries should first be summarized in a **string ,** with each entry being written in a new line. Finally, the **string** will be written into said file.

Hint: Use the **System.lineSeparator()** method to start a new line in a **string**
with.

Note: A method from the **IOOperations** class might be useful for creating the file .

Note: You should catch any exceptions from the **highscore** collection .

If you did everything correctly, the high score file should look like Listing 1.

Listing 1: Sample entries in the "highscores.hs" file.

```
07.10.2018 07 :40 ;205 ;6999.0
20.05.2019 18 :35 ;110 ;3123.0
08.03.2019 22 :13 ;20 ;4875.0
```

5 Cheats (1 + 0.5 + 3 + 3 + 3 = 10.5 points)

In this task, the transfer parameters do not have to be checked. In addition, the costs of the cheat must be deducted from the player's account for each successful execution. In this case, successful execution means that there are enough points available when executing the cheat. The **getCheatCost()** method can be used to determine the price.

Note: A cheat should also deduct points from the score if it does not bring a meaningful solution. (So, for example, there is no longer a valid pair of fields.)

Note: The class can only be used in a meaningful way when all subtasks have been implemented. All cheat IDs and cheat costs can be found in the consts class in the util package.

Note: Keep in mind that cheats can only be performed once you have scored enough points in the game.
To make debugging easier, you can set the **START_POINTS** constant in the **Consts** class to a higher value so that the cheats can be activated. However, don't forget to set the value back to **0** before submitting the project!

a) Write the private static method **boolean isCheatPossible(final Grid grid, final int cheatID).** This should return as a truth value whether the given cheat is currently possible in the given **grid .** To do this, use the values from the consts class.

Note: At this point it is required that the cheat execution option is given, provided the current score is large enough. It is not necessary to check the grid for valid field pairs!

b) Next we want to implement the private static method **List<IToken> shuffle(List<IToken> list) .** This should randomly shuffle a passed list of **ITokens** and then return them.

c) Then implement the private static method **List<IToken> findTokensWithType(final IToken token, final grid grid).** The **grid** should be searched for all **ITokens** that have the same type (display value/DisplayValue) as the **IToken** passed. All **ITokens** found should be returned in the form of a mixed **ArrayList .** The **shuffle()** method can be used for shuffling .

Note: Note that the passed **IToken** itself should **not** be included in the list. **ITokens** that have already been redeemed should not be added to the return list either.

d) Write the public static method **void solvePair(final Grid grid).** The cheat should find a currently solvable pair in the given **grid** and solve it for the player. To find a solvable pair, you can use the **findValidTokens(final Grid grid)** static method . Use a grid method to deselect all currently selected **tokens** and use another grid method to select the found **ITokens** instead . As a last step, you should then reduce the score of the player in the **grid** by the corresponding value, whereby five points should **not** be awarded again for solving a matching pair of fields. (So only the score for the cheat will be deducted.)

Note: This method should always deduct exactly **20** points (even if no pair is solved) if it can be carried out thanks to a sufficient score.

e) Now implement the public and static method **void findPartner(final Grid grid).**
This cheat looks for a detachable partner for a clicked card. If no solvable partner is found, a random partner with the same type on the **grid** is chosen.

Proceed as follows: First
check whether exactly one **IToken** is currently selected in the **grid .** If this is not the case, the method should be terminated. Then use your implementation of **findTokensWithType(final IToken token, final Grid grid)** to find all **ITokens** in **the grid** that can be partners for the selected **IToken .** Check all of these possible partners with the pre-implemented static method **solvable(IToken token1, IToken token2, Grid grid).** The method then returns **true** if the transferred **ITokens** can be solved in the transferred **grid .** If a valid pair is found, the partner can be selected in the **grid** and the score reduced accordingly. (At this point, no points are awarded for correctly solving the pair.) If no solvable combination is found, a non-solvable combination should be chosen in the grid and the score should also be reduced **by** the cheat costs **and** the costs for an incorrect pair selection.

Note: The method should therefore subtract either **10** or **15** points (depending on whether a solvable partner was found), provided there is a large enough score at the beginning .

6 Formatting and Commenting (deduct up to 4 points)

Well-formatted and commented program code helps to make the program easier to understand – including for the proofreader. We expect you to…

- ...describe each method you edit in detail with Javadoc comments (this includes **@param** for each parameter and **@return** for non-void methods),
- …describe how non-trivial code sections work with simple comments (**/* */** or //) ,

- …stick to the standard indentation style, ie child blocks (such as methods, loops, be ding statements) indent with tab or space,
- …use meaningful variable names.

If you do not follow these rules, we will deduct up to **four points** from the evaluation.

Note: You can comment in German or English - decide on one variant!

7 Testing

Test your methods extensively with the techniques presented in the lecture and exercise. Check the methods with invalid parameters and cover any special cases.

We wish you lots of fun and success!