Task 2: GameOfLife CLI

John Horton Conway's Game of Life is no ordinary game, but a mathematical system of two-dimensionally arranged cellular automata. It is the best known example of cellular automata. Conway first published the game in Scientific American 223 in October 1970:

- Task is to implement the game Game Of Life that is interactively playable on a shell.
- Use the given template and add all missing functionality.

Game of Life Rules

The game board is divided into rows and columns. On each square is a cell, which can be either alive or dead. Each cell has 8 neighbors with which it interacts. After each time step ('mutation'), the state of the cell changes as follows:

1. Living cells with less than two neighbors die of loneliness in the following generation.
2. Living cells with more than three neighbors die of overpopulation in the subsequent generation.
3. Living cells with exactly two or three living neighbors remain alive in the subsequent generation.
4. Dead cells with exactly three living neighbors are reborn in the subsequent generation.

In the beginning, the playing field is populated with an initial population. Afterwards, the rules mentioned above are applied simultaneously to all cells for each step in time. This results in complex patterns that remain either stable, oscillate, grow or shrink depending on their shape. Some even wander across the playing field, so-called *spaceships*.

Theoretically, the playing field is infinitely large. However, since memory is limited, a limited playing field is assumed right from the start. Cells outside the playing field are considered dead.

Common initial populations

Below you can see some known initial populations (black cells are alive, white cells are dead):

Common initial populations

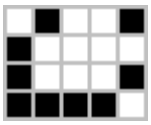
Block (stable) 

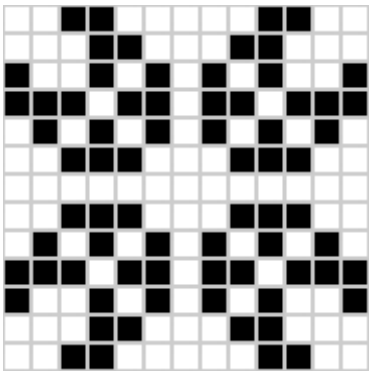
Boat (stable) 

Blinker (two-phase oscillator) 

Toad (two-phase oscillator) 

Glider (spaceship) 

Spaceship (spaceship) 

Pulsar (three-phase oscillator) 

Other known structures are listed in the *Life Lexicon*.

Command-Line Interface

The Game of Life is controlled via a shell. The prompt shall be “gol> “. The following commands need to be supported:

- NEW x y

  Starts a new game with a field size of x columns and y rows.

- ALIVE i j

  Sets the cell in i-th column and j-th row alive.

- DEAD i j

  Sets the cell in i-th column and j-th row dead.

- GENERATE

  Calculates the next generation according to the rules of the game. The (consecutive) number of the generation must be displayed.

- PRINT

  Prints the game field row by row. Living cells are printed with ‘X’ and dead ones with ‘.’.

- CLEAR

  Kills all living cells and resets the generation counter to 0.

- RESIZE x y

  Resizes the game field to x columns and y rows. Living cells whose coordinates are on the new playfield remain alive. The generation counter is not reset.

- SHAPE s

  Loads a predefined initial population with name s. At least the above mentioned common populations must be present and referenceable with their above mentioned name. A CLEAR is automatically performed prior to this. The initial populations are centered on the playfield. If the population does not fit on the playfield an error message is printed.

- HELP

  Prints a meaningful help text.

- QUIT

  Quits the program.

The first letter is sufficient to recognize the command. Commands and shapes are not case sensitive. If the user command is invalid, the program must display a helpful error message in the format `Error! <Message>`:

For unknown commands, the following error message is displayed: `Error! Invalid command.`. For all invalid arguments not mentioned in the commands above (for example too few arguments), the following error message is displayed: `Error! Invalid arguments: EXPLANATION` where 'EXPLANATION' is a placeholder for an explanation of the issue.

Implementation Tips

Files To Submit

The name of your main file (and also of the class!) must be *Shell.java*. This also contains the main method, which can be executed from the command line.

In addition, a text file *Tests.txt* is also required, that again contains a description of your test cases. The test cases should show that you have tested your program extensively.

Generation Change

In Game of Life all births and deaths happen simultaneously in a generation change. In a next time step newly born or freshly deceased cells must not have any influence on the survival of this new generation of a cell. Mistakes in the implementaion can easily happen here!

The following procedure can be used for the generation change:

1. Calculate and store the number of neighbors for each eligible member of the new generation: * Set the neighbor counters of the living cells to zero, otherwise living

cells will not be updated. * Cycle through all currently living cells and add 1 to each of their neighbors' (up to 8) neighbor counters. * All cells whose counter has been updated can belong to the next generation.
2. Pass through all cells of the old population and bury the dead.
3. Pass through all cells of the potential new generation (from 1.) and set the newborns to alive.

Since newborns do not play a role in dying or continuing to live, they should not be counted with current neighbors. Therefore, they should be stored separately.

With the procedure shown above it is possible to calculate the generation change in-place, i.e. on the current playing field. However, alternative procedures in the implementation for calculating the generations are possible and also allowed. You are free in your decision here.

Interface Definition

The use of the *Grid.java* interface for the model is mandatory! The shell must communicate with the model only by means of this interface. The logic regarding the shell and game must be strictly separated.

```
public interface Grid {

  boolean isAlive(int col, int row); // get status of cell

  void setAlive(int col, int row, boolean alive);  // set status of cell

  void resize(int cols, int rows);   // resize grid

  int getColumns();                  // x-dimension

  int getRows();                     // y-dimension

  Collection<Cell> getPopulation();  // all living cells

  void clear();                      // kill all cells

  void next();                       // compute next generation

  int getGenerations();              // get number of generations

  String toString();                 // get string representation

}
```

The class *GridTest.java* contains JUnit tests (JUnit 5), which can be used to check whether your implementation of the *Grid.java* interface fulfills the requirements. The class must be adapted by you at the marked point, but further modifications are not allowed. When submitting, make sure that you also submit this class.

Attention: It is possible that all tests are passed successfully without the requirements being completely fulfilled. Successful tests are necessary, but not sufficient for the evaluation of your submission!

Further Information

- Resizing the playfield should not delete the current living population. Cells remain in the place where they were before. Cells that fall out of the playfield after resizing are removed.
- It is not necessary to implement your own exceptions.
- In addition, always make sure that the state of your playfield is consistent.

Examples

```
gol> NEW 10 10
gol> ALIVE 1 0
gol> ALIVE 2 1
gol> ALIVE 0 2
gol> ALIVE 1 2
gol> ALIVE 2 2
gol> PRINT
.X........
..X.......
XXX.......
..........
..........
..........
..........
..........
..........
..........
gol> GENERATE
Generation: 1
gol> GENERATE
Generation: 2
gol> PRINT
..........
..X.......
X.X.......
.XX.......
..........
..........
..........
..........
..........
..........
gol> DEAD 2 3
gol> GENERATE
Generation: 3
gol> PRINT
..........
.X........
..X.......
.X........
..........
..........
..........
..........
..........
..........
gol> RESIZE 15 15
gol> SHAPE Pulsar
gol> PRINT
...............
```

```
...XX.....XX...
....XX...XX....
.X..X.X.X.X..X.
.XXX.XX.XX.XXX.
..X.X.X.X.X.X..
...XXX...XXX...
...............
...XXX...XXX...
..X.X.X.X.X.X..
.XXX.XX.XX.XXX.
.X..X.X.X.X..X.
....XX...XX....
...XX.....XX...
...............
gol> GENERATE
Generation: 1
gol> GENERATE
Generation: 2
gol> PRINT
....X.....X....
....X.....X....
....XX...XX....
...............
XXX..XX.XX..XXX
..X.X.X.X.X.X..
....XX...XX....
...............
....XX...XX....
..X.X.X.X.X.X..
XXX..XX.XX..XXX
...............
....XX...XX....
....X.....X....
....X.....X....
gol> QUIT
```

Implementation Tips

- Build the program incrementally. Start with the minimum requirements of the project: Commands ‘NEW’, ‘ALIVE’, ‘DEAD’, ‘CLEAR’ and ‘PRINT’.

  Once you have tested that these commands work reliably and when you are happy with your code, continue with the commands that build on that: ‘GENERATE’ and ‘RESIZE’.

  Finally, implement the command ‘SHAPE s’ and any other missing commands.

- Do **not** implement for speed, but for understandability! Make the code easy to understand and work reliably. Introduce classes to keep your data in an understandable way and try to mimic the game’s world with your classes.
- Only care about performance bottlenecks as a last step. Regarding speed, we only care about asymptotic performance issues, i.e., unfitting data structures and nested loops that could be avoided.

General Requirements

- You can modify the behavior of any method of the template, and add public methods if necessary.
- BUT: Do not modify any public method signatures or interfaces. You are not allowed to add method parameters to existing public methods.
- The output and return values of implemented methods must exactly match the output described here. Otherwise, tests will fail.
- Documentation of the source code is part of the task. The Google Java Style Guide must be fulfilled.
    - Setup in Intellij can be done as follows:

      Download the *google_checks.xml* to a location of your choice and import it next in Intellij. This is done by opening the settings (`Menu` → `Settings`), and then navigating to `Editor` → `Code Style` → `Java`. Finally, click on the gearwheel and afterwards on `import Scheme` → `CheckSytle Configuration`. There, add the downloaded xml file from before and apply the changes.

- The asymptotic run-time performance of the implementation is part of the task.
- Your solution must be anonymous. Your name may not appear in the solution.
- Your solution may not contain compiled data (no `build` directory or `.class` files). Please check the content of your ZIP file before uploading.

Helpful Resources

- Data structures more flexible than arrays:
    - java.util.List, especially java.util.ArrayList
    - java.util.Map, especially java.util.HashMap

While reading JavaDoc, you may encounter the following types that could be confusing:

- Stream is an advanced data structure similar to lists. It can be consumed only once and has lots of methods for transformation and filtering. Often uses method references and other advanced topics. You can turn any Stream into a *List*-object like this:
- `Stream<Integer> numberStream = // .. snip ..;`
- `List<Integer> numberList = numberStream.collect(Collectors.toList());`

Good luck!