Gomoku (aka, five in a row) game

The goal of the exercise is to implement an engine to play Gomoku (https://en.wikipedia.org/wiki/Gomoku). This game can be played in square boards of different sizes (historically, 15x15 or 19x19, but the implementation should be flexible enough to accommodate arbitrary squared dimensions as well). Players alternate turns placing a cross or a circle in a cell. The winner is the first player to form an unbroken chain of five stones horizontally, vertically, or diagonally.

All the classes needed to implement this game are in `it.unive.dais.po1.exercise2` package. Since the interfaces of Gomoku and Tic-tac-toe are the same, the package contains two abstract classes:

- Class `Player` storing the mark of the player (represented as a `Mark` field `mark`) providing:
    - a constructor receiving as parameter the mark of the player (either cross or circle, represented as instance of `Mark`), and
    - an abstract method `play` that, given a board, tries to play a move (that is, it marks one of the cells with his/her mark if the board is not full and there is not yet a winner), and returns true if he/she was able to play his/her round. Note that there is no assumption about how the player plays the game (e.g., randomly, taking the "first" empty cell, considering what is already on the board, etc..), but the player should always play his/her move if possible.
- Class `Board` storing the status of the board (represented as a `Mark[][]` matrix in field `board`) and providing:
    - a constructor receiving the size of the square board,
    - `putMark` method that given a `Mark`, and the coordinates x and y of a cell in the board, returns false if there is already a winner or the cell is not empty, or otherwise puts the mark in the given cell returning true,
    - `getMark` method that given the coordinates x and y of a cell returns the mark in the cell, or null if the cell is empty,
    - `getDimension` method that returns the dimension of the square board,
    - `isFull` method (returning true if and only if all the cells of the board are not empty), and
    - `winner` abstract method that returns the mark of the winner, or null if there is no winner yet.

  The coordinates of cells in the board ranges from 0 to `dimension-1` (e.g., cell at (0, 0) is the upper left cell, while (dimension-1, dimension-1) is the lower right cell) where dimension represents the dimension of the square board (e.g., 3 if we would play tic-tac-toe).

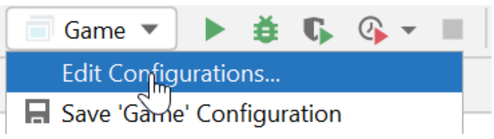
You can find all the classes and tests of this exercise in the GitHub repository of the course (https://github.com/pietroferrara/PO1_2020).

In particular, class `Mark` represents a mark (that might be either a cross or a circle) in the tic-tac-toe board. The implementation of this class is given: it contains two static public methods, `getCircle` and `getCross`, that returns the instance of `Mark` class representing a circle and a cross, respectively.

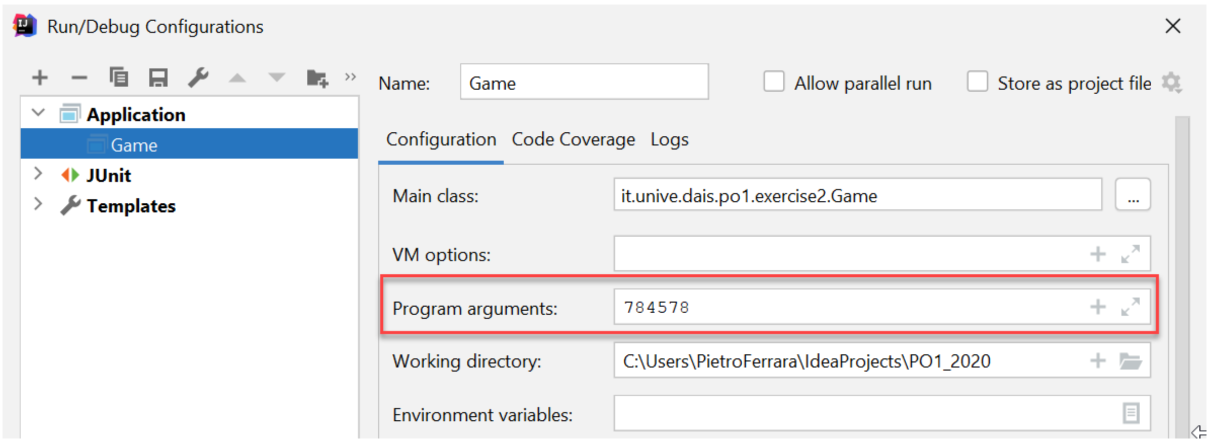
You are asked to extend these two classes in package `it.unive.dais.po1.exercise2.m<your_matricola>` (note that the last part of the package's name starts with `m` followed by your matricola) with classes `GomokuBoard` and `Player` (note that you can keep the same name of the extended class in package `it.unive.dais.po1.exercise2` since they belong to two different packages. Method `play` of class `Player` should be able to play both Gomoku and Tic-tac-toe games (for the latter you can copy and paste the code you wrote for exercise 1). In addition, you should modify `TicTacToeBoard` (developed in Exercise 1) to extend class `Board`, and move it to package `it.unive.dais.po1.exercise2.m<your_matricola>`.

Note that class `GomokuGame` provides some utilities to debug the implementation: method `playGame` plays a game end-to-end and returns the mark of the winner (printing in the console all the rounds of the game), `print` prints in the console the status of a board (representing crosses by X, circles by O, and empty cells by e), while `main` simply plays a full game. Therefore, class `Game` can be used to run the whole implementation and debug it. In order to run the main method, you need to specify through program arguments your matricola. You can do it by

1) editing the run configuration of Game as follows



2) add the matricola number in the "Program arguments" field as follows


Run/Debug Configurations

Name: Game

Allow parallel run

Store as project file

Configuration | Code Coverage | Logs

Main class: it.unive.dais.po1.exercise2.Game

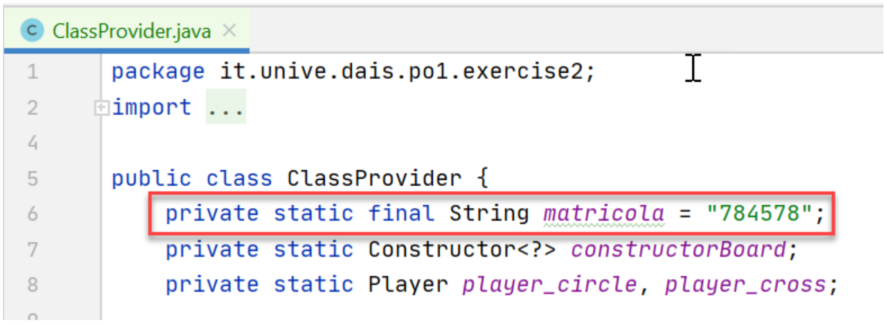
VM options:

Program arguments: 784578

Working directory: C:\Users\PietroFerrara\IdeaProjects\PO1_2020

Environment variables:

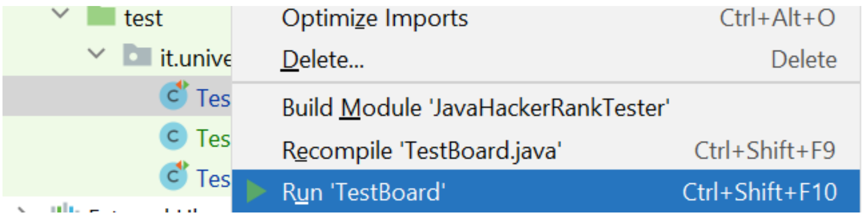
You can self-evaluate your exercise with the tests that are in the `/test` directory. Before running the test, you need to manually specify your matricola number in field `matricola` of `ClassProvider` class:



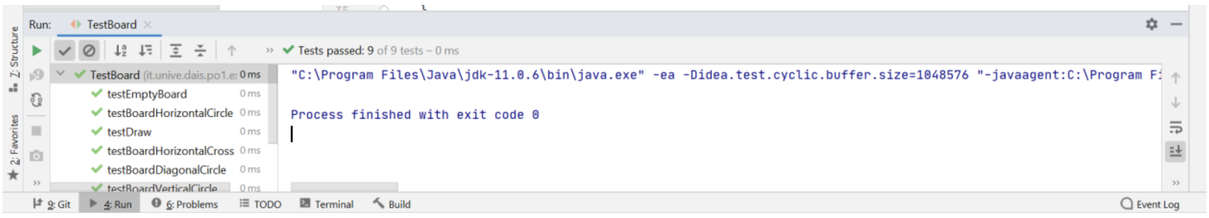
```java
package it.unive.dais.po1.exercise2;
import ...

public class ClassProvider {
    private static final String matricola = "784578";
    private static Constructor<?> constructorBoard;
    private static Player player_circle, player_cross;
```

In particular, the tests are structured into three distinct sets of tests: one to evaluate the functionalities of the board, one for the player, and one checking if the structure is correct (e.g., contains the fields and methods expected, there is no unneeded public method, etc..). The tests are implemented as JUnit tests. You can run them by right-clicking on the class file, and selecting “Run ’<..>Test’”:


test
it.unive
Tes
Tes
Tes
Optimize Imports Ctrl+Alt+O
Delete... Delete
Build Module 'JavaHackerRankTester'
Recompile 'TestBoard.java' Ctrl+Shift+F9
Run 'TestBoard' Ctrl+Shift+F10

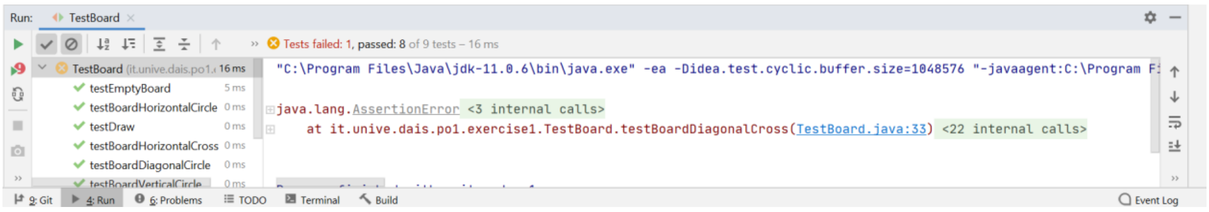
The results of the tests are then visualized in the lower part in the Run view:


Run: TestBoard
Tests passed: 9 of 9 tests – 0 ms
TestBoard (it.unive.dais.po1.e: 0 ms
testEmptyBoard 0 ms
testBoardHorizontalCircle 0 ms
testDraw 0 ms
testBoardHorizontalCross 0 ms
testBoardDiagonalCircle 0 ms
testBoardVerticalCircle 0 ms
9: Git
4: Run
6: Problems
TODO
Terminal
Build
Event Log

```
"C:\Program Files\Java\jdk-11.0.6\bin\java.exe" -ea -Didea.test.cyclic.buffer.size=1048576 "-javaagent:C:\Program Fi

Process finished with exit code 0
```

If a test caused a runtime error it is marked as red, and if it did not pass one of the checks of the test (e.g., the board didn’t return the expected winner) it is marked as orange. Selecting the failing tests, you can visualize the line of the test that failed:


Run: TestBoard
Tests failed: 1, passed: 8 of 9 tests – 16 ms
TestBoard (it.unive.dais.po1.(16 ms
testEmptyBoard 5 ms
testBoardHorizontalCircle 0 ms
testDraw 0 ms
testBoardHorizontalCross 0 ms
testBoardDiagonalCircle 0 ms
testBoardVerticalCircle 0 ms
9: Git
4: Run
6: Problems
TODO
Terminal
Build
Event Log

```
"C:\Program Files\Java\jdk-11.0.6\bin\java.exe" -ea -Didea.test.cyclic.buffer.size=1048576 "-javaagent:C:\Program Fi

java.lang.AssertionError <3 internal calls>
    at it.unive.dais.po1.exercise1.TestBoard.testBoardDiagonalCross(TestBoard.java:33) <22 internal calls>
```