Playing tournaments

In the previous exercises, we have implemented a board and a player for Tic-Tac-Toe and Gomoku games. Now it is time to leverage this environment to play tournaments!

All the classes needed to implement this game are in `it.unive.dais.po1.exercise3` package. Some classes are rather similar to the ones you have seen in the previous exercise with some slight differences. In addition, few classes to represent games and tournaments have been added:

- Class `GameException` representing a checked exception happening when something goes wrong while playing a game;
- Class `Player` providing:
  - an abstract method `play` that, given a board and a mark (note that this has been added in this exercise, previously it was a parameter of the constructor), plays 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). If it was unable to play, it raises a `GameException`
  - an abstract method `getIdentifier` returning the identifier of the player represented as a `String` object;
- Class `RandomPlayer` extending Player and implementing a player that puts his mark in a random place. This class is provided for your convenience to build up a tournament with your implementation of player (developed during the previous exercise) and at least another player.
- Class `Mark` representing a mark that can be put in the board (circle or cross)
- Class `Result` representing the result of a game (the first player won, the second player won, or draw)
- Final class `Game` that receives two players and a board, and provides a method `play()` that plays the game and returns the result
- 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
  - `clone` public abstract method (returning a copy of the current board),
  - isValidMove abstract method (that given the previous board and the mark used to play a mode returns true if and only if the move was legal), and
  - `winner` abstract method that returns the mark of the winner, or null if there is no winner yet.
- Abstract class `Tournament` representing a tournament. Its constructor receives a collection with the players involved in the tournament, and number of point that should be given for a draw or a won game. It provides a `play()` method that plays the whole

tournament (relying on method `Game.play()` to play single games) and returns the final standings.

- Abstract class `Standing` computing and representing the standing of a tournament. It defines three methods:
    - `recordResult` that given two players and a result, store the result
    - `getPointsOfPlayer` that returns the number of points obtained by a given player so far
    - `getStanding` that returns the players in the order of the standing (e.g., at index 0 there is the player that is first in the ranking).
-

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).

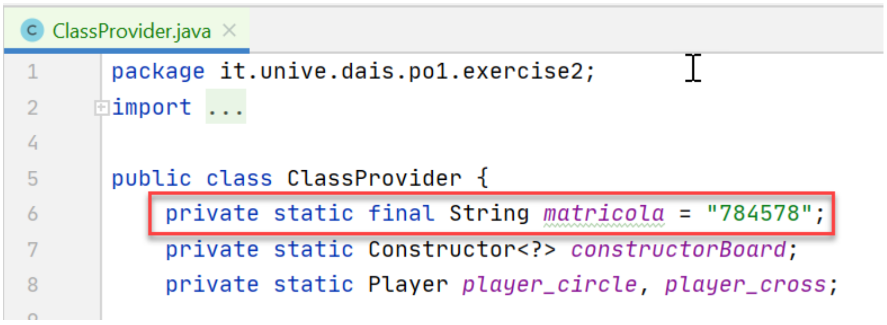
You can find all the classes and tests of this exercise in the GitHub repository of the course (https://github.com/pietroferrara/PO1_2020).

You are asked to:

- extend classes `Board` and `Player` in package `it.unive.dais.po1.exercise3.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.exercise3` since they belong to two different packages.
- Extend abstract class `Standing` with your implementation of the standing and providing the implementation of all the abstract methods
- Extend abstract class `Tournament` implementing its method `play()`. Your implementation of class Standing will be the base for the standings this method will return. Note that that you are free to implement the type of tournament you prefer (e.g., knockout or group tournaments), and even more than one.
- Optional: you are free to modify class Game to dump information about the game, e.g., as text (as I have done for your tournament), XML files (as we have seen in the lecture about annotations), …

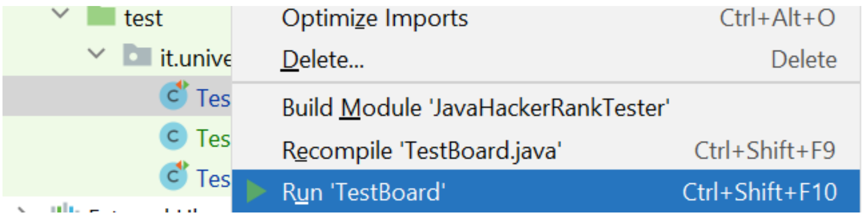
You can self-evaluate a part of your exercise with the tests that are in the `/test` directory. The tests cover only few classes (in particular, your implementation of the player and of the board) and few functionalities. Since building up the tournament and playing games needs to be manually configured, this part is left out from the tests, but you are strongly encouraged to test this part of the exercise by yourself.

Before running the test, you need to manually specify your matricola number in field `matricola` of `ClassProvider` class:



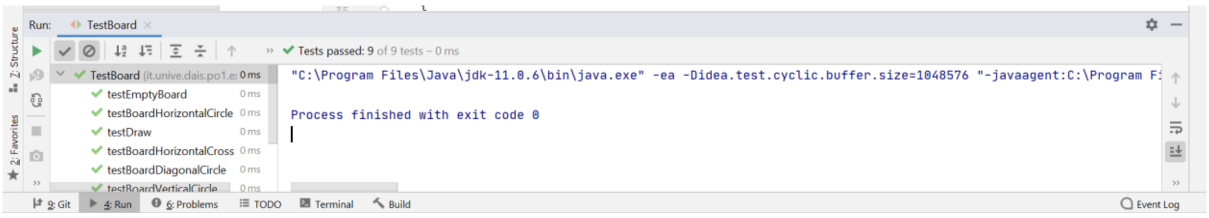
```java
package it.unive.dais.po1.exercise2;
import ...

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

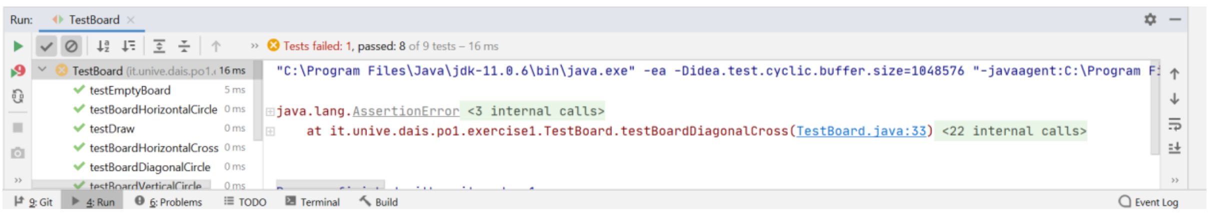
The results of the tests are then visualized in the lower part in the Run view:



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>
```