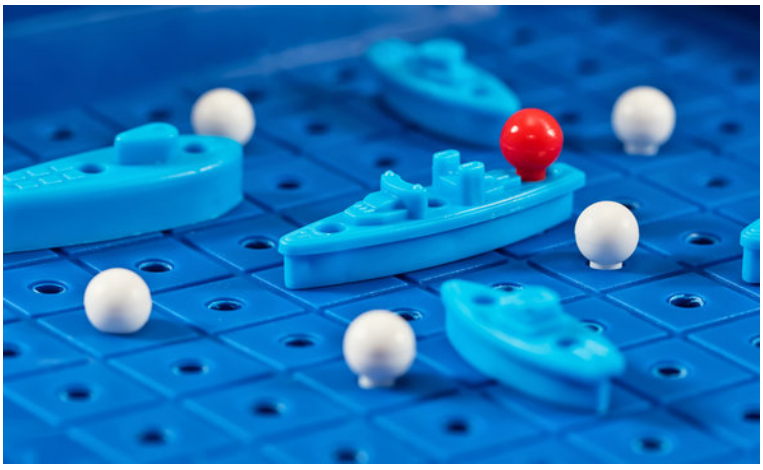
NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Department of Computer Science

Introduction to Programming
Battleship

First Programming Project



Academic Year 2023/24

Version 1 (2023/10/12)

Preamble

The programming project is developed **individually** and is submitted to Mooshak, which accepts submissions until **8:00 pm** on **October 27, 2023**. Read this document with the utmost attention, to understand the problem very well and all the details about the Mooshak contest and the evaluation criteria for the programming project

1 Concepts and Goal of the Programming Project

Who has never played Battleship? Each of the two players begins by placing their ships on a grid, hidden from the opponent. When it is their turn to play, the player takes shots to hit the opponent's ships. The goal is to sink all of the opponent's ships before the opponent sinks theirs.

In this version of the game, some rules are different from the usual ones. The battle takes place on linear *grids*, which are divided into *cells*, as illustrated in Figure 1. The cells are identified by their order number, which is assigned from left to right (i.e., the first cell or the cell in the first position, the second cell or the cell in the second position, etc.). The *size* of a grid is the number of cells in the grid.

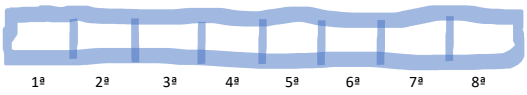


Figura 1: Grid with 8 cells

Before the game begins, the players agree on the size of the grids they will use and the composition of their fleets (which types of ships and how many of each type). Then, each player arranges their ships on their grid without overlaps. Note that ships can be placed side by side, as shown in the arrangement schematised in Figure 2. In this example, the fleet consists of three ships: a three-cell ship in the first, second, and third cells; a two-cell ship in the fourth and fifth cells; and a one-cell ship in the seventh cell.

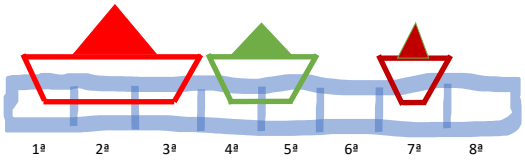


Figura 2: Fleet Arrangement

The players take turns playing. Each turn consists of taking a shot at a position on the opponent's grid. If the shot hits a ship, that ship *sinks* (i.e., ceases to exist), regardless of the number of cells it has. If the shot hits a position where a ship used to exist, the player incurs a penalty. A player's *score* is determined by the ships they sank and the shots they fired at cells with "sunk ships". The game ends as soon as one player's entire fleet is sunk.

The arrangement of the fleet on the grid will be represented by a sequence of characters whose length is equal to the size of the grid. When a cell does not contain a ship, the character in the sequence is '.' (full stop). Cells where a ship is present have the same letter in corresponding positions of the sequence, and if there are ships side by side, the letters representing them are different. Note that any arrangement can be represented by many different sequences. For example, "`RRRGG.R.`" and "`AAABB.C.`" are two representations of the arrangement shown in Figure 2. If, at a certain point in the game, there are sunk ships, the representation of the *fleet state* has the character '*' in the positions where those ships were located. Let's look at three examples, assuming that the fleet's arrangement had been described as "`RRRGG.R.`".

- If the only sunk ship was the two-cell one, the fleet's state would be represented as "`RRR**.R.`".
- If the only ship not sunk was the one-cell one, the fleet's state would be "`*****.R.`".
- If the entire fleet had already been sunk, its state would be "`*****.*.`".

The goal of this programming project is to program, in Java, a version (significantly modified) of the Battleship game. With information about the players' names, the arrangement of their fleets, and the sequence of shots fired, the program should be able to indicate, at any point in the game, the score and fleet status of each player. If the game has not yet ended, the program should also determine which player is next in line to take a shot.

2 Game Rules

Before the game begins, the two players agree on the grid size to use, the composition of their fleets, and who goes first. Then, each player arranges their ships on their grid (without overlapping). Players take turns, one at a time.

At the start of the game, both players have zero points. As long as the game has not ended, the player with the right to play takes a shot at a position on the opponent's grid, resulting in one of the following three situations:

- If there is a ship at that position, the ship sinks (ceases to exist). The player earns as many points as the number of cells of the ship, multiplied by 100.
- If there was never a ship at that position, the player's score remains unchanged.
- If there is no ship at that position because the existing ship was sunk, the player loses as many points as the number of cells of the ship that used to exist, multiplied by 30.

The game ends when one of the players' fleets is completely sunk. From that moment on, no player can take shots; consequently, the scores and fleet states of the players can no longer change. The *winner* is the player with the highest score when the game ends; if both have the same number of points, the one who has sunk the opponent's entire fleet wins. Note that the score can be positive, zero, or negative.

3 System Specification

The application interface should be simple so that it can be used in different environments and facilitate the testing process automation. For these reasons, the input and output must adhere to the precise format specified in this section. You can assume that the input complies with the value and format constraints indicated, meaning that the user does not make errors beyond those foreseen in this document.

The program reads lines from the standard input (`System.in`), writes lines to the standard output (`System.out`), and is case sensitive (for example, the words "quit" and "Quit" are different).

Input Format

The input has the following structure (where the symbol ↩ represents the newline character):

playerName$_1$↩
fleetArrangement$_1$↩
playerName$_2$↩
fleetArrangement$_2$↩
command↩
command↩
.....
command↩
quit↩

where:

- *playerName*$_1$ and *playerName*$_2$ are two different sequences of characters, each with lengths between 1 and 40 characters. These sequences may consist of multiple words, such as "John Doe";
- *fleetArrangement*$_1$ and *fleetArrangement*$_2$ are two sequences of characters with the same length (which is a number between 2 and 100). Each element in the sequence is either an uppercase letter (from 'A' to 'Z', without accents or cedillas) or the character '.' (period), and they must contain at least one letter;
- *command* is one of four commands, (called *player-command*, *score-command*, *fleet-command*, and *shot-command*) or an invalid command, explained below.

The first line of the input contains the name of the first player, who starts the game, and the second line describes the arrangement of their fleet. The third and fourth lines specify, respectively, the name and arrangement of the other player's fleet. An arbitrary number of commands follows. The last line contains a special command, the *quit-command*, which can only occur in the last line because it terminates the program's execution.

Player-Command

The player-command indicates that we want to know who is the next player to take a shot, at the current moment in the game. This command does not change the game's state. The lines with player-commands have the following format:

`player`↩

The program writes a line, distinguishing two cases:

- If the game has already over, the line has:

`The game is over`↩

- In the remaining cases, the line has the following format, where *playerName* represents the name of the next player to take a shot:

`Next player:` *playerName*↩

Score-Command

The score-command indicates that we want to know the score of the player with the given name, at the current moment in the game. This command does not change the game's state. The lines with score-commands have the following format (with a space separating the two components):

`score` *playerName*↩

where *playerName* is a non-empty sequence of characters.

The program writes a line, distinguishing two cases:

- If *playerName* is not one of the player's names, the line has:

`Nonexistent player`↩

- In the remaining cases, the line has the following format, where *score* represents the score of the player referred to in the command:

playerName `has` *score* `points`↩

Fleet-Command

The fleet-command indicates that we want to visualise the state of the fleet of the player with the given name, at the current moment in the game. This command does not change the game's state. The lines with fleet-commands have the following format (with a space separating the two components):

`fleet` *playerName*↩

where *playerName* is a non-empty sequence of characters.
The program writes a line, distinguishing two cases:

- If *playerName* is not one of the player's names, the line has:

  `Nonexistent player`↩

- In the remaining cases, the line has the following format, where *fleetStatus* represents the state of the fleet of the player referred to in the command:

  fleetStatus↩

  The state of the fleet corresponds to the arrangement of the player's fleet where all the letters representing ships that have been sunk by the opponent's shots have been replaced by the character '*' (asterisk).

Shot-Command

The shot-command indicates that, at the current moment in the game, the player who has the right to play has taken a shot on the opponent's grid and which cell the player chose. The lines with shot-commands have the following format (with a space separating the two components):

`shoot` *position*↩

where *position* is an integer number.
The program should use this information to update the game's state but should not write any results except in the following two cases, where the game's state does not change, and the program writes a line:

- If the game is already over, the line has:

  `The game is over`↩

- If the game is not over yet, but *position* is not a number between 1 and the size of the grid, the line has:

  `Invalid shot`↩

Quit-Command

The quit-command indicates that we want to terminate the execution of the program.

`quit`↩

The program ends, writing a line. Two cases are distinguished:

- If the game is not over yet, the line has:

  `The game was not over yet...↩`

- If the game is already over, the line has the following format, where *winningName* denotes the name of the player who won the game:

  winningName `won the game!↩`

Invalid Commands

Whenever the user writes a line that does not start with the words "player", "score", "fleet", "shoot", or "quit", the game state must not be changed, and the program must write a line with:

`Invalid command↩`

4 Examples

Here are some examples. The left column illustrates the interaction: the input is written in blue, and the output is in black. All input and output lines end with a newline symbol, which has been omitted for readability. The right column provides information for the reader, serving only as a reminder of the rules described earlier. In this column, "**c**" abbreviates "cells(s)" and "pts" abbreviates "points".

Example 1

`John`	Name of the player who starts the game.
`RRRGG.R.`	Jonh's fleet arrangement.
`Doe`	Name of the other player.
`DD.U.TTT`	Doe's fleet arrangement.
`player`	
`Next player: John`	
`shoot 8`	John hits a 3 **c** ship; wins 300 pts.
`fleet Doe`	
`DD.U.***`	
`score John`	
`John has 300 points`	
`shoot 6`	Doe hits the water.
`shoot 6`	John hits a sunk ship (of 3 **c**); loses 90 pts.
`shoot 8`	Doe hits the water.
`fleet John`	
`RRRGG.R.`	
`shoot 3`	John hits the water.
`shoot 7`	Doe hits a 1 **c** ship; wins 100 pts.
`shoot 1`	John hits a 2 **c** ship; wins 200 pts.
`score John`	
`John has 410 points`	
`shoot 5`	Doe hits a 2 **c** ship; wins 200 pts.
`quit`	
`The game was not over yet...`	

Example 2

`007 The Spy`	Name of the player who starts the game.
`...X...XX...`	007 The Spy's fleet arrangement.
`Alice`	Name of the other player.
`I.........KK`	Alice's fleet arrangement.
`Score 007`	The command is "score" (all letters are lowercase).
`Invalid command`	
`score 007`	"007" it is not the name of any player.
`Nonexistent player`	
`fleet Alice Zoe`	"Alice Zoe" it is not the name of any player.
`Nonexistent player`	
`shoot 0`	
`Invalid shot`	
`shoot 12`	007 The Spy hits a 2 **c** ship; wins 200 pts.
`shoot 1`	Alice hits the water.
`shoot 11`	007 The Spy hits a sunk ship (of 2 **c**); loses 60 pts.
`shoot 2`	Alice hits the water.
`shoot 12`	007 The Spy hits a sunk ship (of 2 **c**); loses 60 pts.
`shoot 3`	Alice hits the water.
`shoot 11`	007 The Spy hits a sunk ship (of 2 **c**); loses 60 pts.
`shoot 13`	
`Invalid shot`	
`player`	
`Next player: Alice`	
`shoot 8`	Alice hits a 2 **c** ship; wins 200 pts.
`score 007 The Spy`	
`007 The Spy has 20 points`	
`score Alice`	
`Alice has 200 points`	
`fleet 007 The Spy`	
`...X...**...`	
`fleet Alice`	
`I.........**`	
`shoot 1`	007 The Spy hits a 1 **c** ship; wins 100 pts; the game ends.
`shoot -5`	
`The game is over`	
`shoot 6`	
`The game is over`	
`player`	
`The game is over`	
`fleet Alice`	
`*.........**`	
`fleet 007 The Spy`	
`...X...**...`	
`score 007 The Spy`	
`007 The Spy has 120 points`	
`score Alice`	
`Alice has 200 points`	
`quit`	Alice won because has more points than 007 The Spy.
`Alice won the game!`	

5 Programming Project Submission

The programming project is submitted to Mooshak. You must submit a `.zip` file to Problem A of contest **IP2324-P1**. Do not forget that:

- The archive should contain only all the `.java` files that you have created to solve the problem.
- The archive must necessarily contain a `Main.java` file, where the `main` method is.
- The `Main` class must belong to the default package.
- The Java version installed on Mooshak is 17.[1]

Each student will receive access credentials to the IP2324-P1 contest (which should be different from the access credentials for the IP2324-Aulas contest) at their institutional email address.

The IP2324-P1 contest opens on October 16th and closes at **8:00 pm** on **October 27, 2023** (Friday). You can resubmit a program as many times as you like, up to the submission deadline. Only the program that obtains the **highest score** in Mooshak will be evaluated; if there are several programs with the highest score, the **last one** (of those) submitted will be evaluated. If you want the evaluated program to be different, you must send a message to Professora Margarida Mamede (mm@fct.unl.pt) up to one hour after the contest closes, indicating the number of the submission that you want to be evaluated.

6 Evaluation Criteria

According to the FCT NOVA Knowledge Evaluation Regulation:

- There is fraud when:
  (a) You use or attempt to use, in any way, in a test, exam, or other form of in-person or remote evaluation, unauthorised information or equipment;
  (b) You give or receive unauthorized help in exams, tests, or any other component of the knowledge evaluation;
  (c) You give or receive help, not permitted by the rules applicable to each case, in carrying out of practical assignments, reports or other evaluation elements.
- Students directly involved in a fraud are immediately excluded from the course assessment process, without prejudice of a disciplinary or civil procedure.

The document Allowed and non-allowed collaborations, available in Moodle, clarifies the points transcribed above. Students who commit fraud on an assignment will not receive frequency for it.

The evaluation of the programming project has two independent components, whose grades are added to obtain the grade of the programming project:

- **Functionality** (correction of the results produced)**: 12 points**

  A program submitted to the contest that only uses the allowed library classes and that gets P points in Mooshak will be scored $P/10$ points.[2] If the program uses classes from forbidden libraries, the functionality score will be reduced by $P/10$ points.

[1] This information is irrelevant if you only use the Java instructions given in class because, in that case, the program should behave the same on your machine and on Mooshak.

[2] The project assignment can be completed incrementally. For example, you can start by assuming that there are only one-celled ships and that no player shoots at sunken ships. Of course, until the program produces the correct results in all of Mooshak's tests, it will not receive 120 points.

The allowed library classes are **only the ones used in the theoretical-practical classes** that took place until October 11, 2023.

You will be provided with practice tests similar to those used by Mooshak. If your program produces the correct results with all the practice tests, it is likely (but not guaranteed) that you will receive 120 points on Mooshak.

- **Code quality: 8 points**[3]

  A quality code has, among others, the following characteristics:

  - **Several classes that characterise the different entities of the problem well**;
  - Classes, methods, variables and constants with well-defined goals and appropriate access constraints;
  - Simple and well-structured algorithms, implemented with the most suitable instructions;
  - Identifiers that express the concepts they represent, written according to the conventions taught (for example, the name of a class must be a noun that starts with a capital letter);
  - Correct indentation[4], lines with 100 characters (maximum) [5], and methods with 25 lines (maximum);
  - A comment before each method (except for the main method), which briefly explains what the method does.

Good luck!

[3] Notice that code quality has a significant impact on the programming project's grade.
[4] In Eclipse, you can use the Ctrl I command (Windows) or the Command I command (Mac).
[5] When counting the number of characters in a line of code, consider that one tab is equivalent to 4 characters.