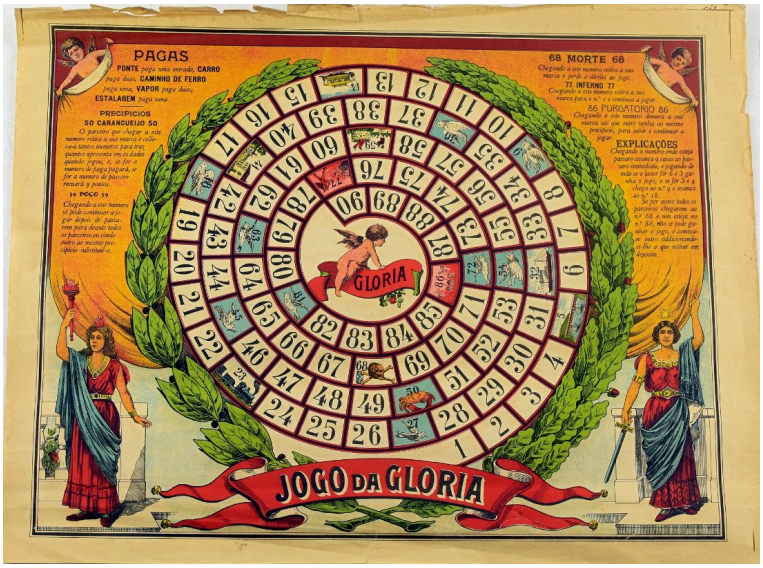
NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Departamento de Informática

Introdução à Programação
The Goose Game (Jogo da Glória)

First Programming Project



Academic Year 2022/23

Version 1 (2022/10/24)

Preamble

The programming project is developed **individually** and is submitted to Mooshak, which accepts submissions until **20:00** on **November 7, 2022**. Read this document with the utmost attention, to understand the problem very well and all the details about the Mooshak contest and the assessment criteria for the programming project.

1 Concepts and Goal of the Programming Project

The Goose Game (*Jogo da Glória*) is a board game that marked the childhood of many Portuguese people. The traditional *board*, shown in Figure 1, is composed of 90 *squares*, numbered from 1 to 90. Each player has a coloured *piece* (as those illustrated in Figure 2), which starts on square 1. When it is his turn to play, the player rolls two dice to determine how many squares the piece advances. Whoever arrives at the 90th square first wins.

Figure 1: Board of the Game

For more excitement, there are special squares, called *fines*, *cliffs* and *geese*. If the piece stops on a fine, the player cannot roll the dice for a number of turns. The penalty suffered when the piece stops on a cliff varies greatly with the cliff. For example, the crab makes the piece move backwards (instead of moving forwards). Unlike fines and cliffs, stopping at a square with a goose is fortunate, because the piece immediately advances 9 more squares.

The goal of this programming project is to program in Java a very simplified version of the Goose Game, in which there are exactly three players. With the information about the colours of their pieces, the board, and the sequence of rolls of the dice, the program must be able to indicate, at any moment of the game, on which square each player's piece is and, if the game is not over yet, who is the next player to roll the dice and whether or not a given player can roll them when it is his turn to play.

Figure 2: Pieces

2 Game Rules

Three players compete against each other, each with a different piece. For simplicity, every player is identified by the colour of his piece and is said to be on a square if his piece is on that square. Players play one at a time. The order in which players play is cyclic and is defined before the game starts.

For example, if the players have pieces with the colours R, G and B (*red*, *green* and *blue*) and this is the order in which they play: the first to play is the player (who has the piece) R; after R plays, it is G's turn to play; after G plays, it is B's turn; after B plays, it is again R's turn to play.

The board has a number C of squares, numbered from 1 to C (with $10 \leq C \leq 150$). When the game starts, all pieces are on square 1 and the goal is to reach square C. Any square has one, and only one, type. That type can be *normal*, *goose*, *fine*, or *cliff*. Geese, fines and cliffs can only occur between squares 2 and $C-1$. In this interval, all "multiple of 9" squares (squares 9, 18, 27, etc., without exceeding $C-1$) have a goose.

When a player is on a square c, rolls the two dice and p spots come out, the player *advances p squares* (where $1 \leq c < C$ and $2 \leq p \leq 12$). This means that the player goes to square $c+p$, unless that square does not exist (because it is larger than C), in which case the player moves to square C. Similarly, a player who is on a square c and who *moves back p squares* goes to square $c-p$, unless that square does not exist (because it is smaller than 1); in that case, the player goes to square 1.

At the beginning of the game, all players are on square 1 and have a penalty of zero moves (they have never been fined).

When it is his turn to play, the player's behaviour depends on the value of his penalty. The move can change the square where he is and the penalty he has.

- If the penalty value is positive, the player cannot roll the dice (having to pass them to the next player). The player remains on the same square and the value of his penalty drops by one.

  These moves are considered to **occur as soon as possible**.

- If the penalty value is zero, the player rolls the dice and his piece advances as many squares as the total number of spots that came out.

  - If the piece stops on a normal square, the player's penalty does not change. If that square is C, the player wins the game.
  - If the piece stops on a goose, the player advances 9 (more) squares. His penalty does not change.
  - If the piece stops on a fine, the player's penalty increases by two units.
  - If the piece stops on a cliff, the player returns to the square where he was (before he rolled the dice) and then moves back as many squares as the total number of spots that came out. His penalty does not change.

The game ends as soon as one of the players reaches square C. At that moment, no player can roll the dice and all pieces remain on the squares where they are.

3 System Specification

The application's interface is intended to be simple, so that it can be used in different environments and allow automating the testing process. For these reasons, the input and the output must respect the precise format specified in this section. You can assume that the input obeys the indicated value and format restrictions, that is, that the user does not make mistakes 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 "exit" and "Exit" are different).

Input Structure

The input has the following structure (where the symbol $\hookleftarrow$ represents the newline character):

colours↩
numSquares↩
M↩
fine_1 fine_2 $\cdots$ fine_M↩
P↩
cliff_1 cliff_2 $\cdots$ cliff_P↩
command↩
command↩
.....
command↩
exit↩

where:

- *colours* is a sequence of three different capital letters (from 'A' to 'Z');
- *numSquares* is an integer between 10 and 150;
- M and P are integers between 1 and one third of *numSquares*;
- $\mathit{fine}_1, \ldots, \mathit{fine}_M$ are M integers in ascending order, separated by a single space;
- $\mathit{cliff}_1, \ldots, \mathit{cliff}_P$ are P integers in ascending order, separated by a single space;
- the numbers $\mathit{fine}_1, \ldots, \mathit{fine}_M, \mathit{cliff}_1, \ldots, \mathit{cliff}_P$ vary between 2 and *numSquares* -1, they are all distinct and none is a multiple of 9;
- *command* is one of four commands (called *player-command*, *square-command*, *status-command*, and *dice-command*) or an invalid command, explained below.

The input first line has the colours of the players, in the order in which they play. The second line has the number of squares on the board. The third and fifth lines indicate how many squares are fines (M) and how many squares are cliffs (P). The fourth and sixth lines specify, respectively, which are the squares where there is a fine and which are the squares where there is a cliff.
An arbitrary number of commands follows. The last line has a special command, the *exit-command*, which can only occur on the last line because it terminates the execution of the program.

Player-Command

The player-command indicates that we want to know who is the next player to roll the dice, at the current moment of the game. This command does not change the game state. Lines with player-commands have:

player↩

The program writes a line, distinguishing two cases:

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

  The game is over↩

- In the remaining cases, the line has the following form, where *playerColour* stands for the colour of the next player to roll the dice:

  Next to play: *playerColour*↩

Square-Command

The square-command indicates that we want to know on which square the player with the given colour is, at the current moment of the game. This command does not change the game state. Lines with square-commands have the following form (with a single space separating the two components):

`square` *playerColour*↩

where:

- *playerColour* is a non-empty sequence of characters.

The program writes a line, distinguishing two cases:

- If *playerColour* is not the colour of one of the players, the line has:

`Nonexistent player`↩

- In the remaining cases, the line has the following form, where *playerSquare* stands for the square where the player referred to in the command is:

playerColour `is on square` *playerSquare*↩

Status-Command

The status-command indicates that we want to know whether, at the current moment of the game, the player with the given colour can or cannot throw the dice, when (and if) it is his turn to play. This command does not change the game state. Lines with status-commands have the following form (with a single space separating the two components):

`status` *playerColour*↩

where:

- *playerColour* is a non-empty sequence of characters.

The program writes a line, distinguishing four cases:

- If *playerColour* is not the colour of one of the players, the line has:

`Nonexistent player`↩

- If *playerColour* is a player's colour and the game is already over, the line has:

`The game is over`↩

- If *playerColour* is a player's colour, the game is not over yet and the player referred to in the command can roll the dice, when and if it is his turn to play, the line has the following form:

playerColour `can roll the dice`↩

- In the remaining cases, the line has the following form:

playerColour `cannot roll the dice`↩

Dice-Command

The dice-command indicates that, at the current moment of the game, the player who has the right to roll the dice has rolled the dice and how many spots have come out on each dice. Lines with dice-commands have the following form (with a single space separating consecutive components):

`dice` *diceSpots*$_1$ *diceSpots*$_2$↩

where:

- *diceSpots*$_1$ and *diceSpots*$_2$ are integer numbers.

The program must use this information to update the game state, but it must not write any results, except in the following two cases, in which the game state does not change and the program writes a line:

- If *diceSpots*$_1$ or *diceSpots*$_2$ is not a number between 1 and 6, the line has:

  `Invalid dice`↩

- If *diceSpots*$_1$ and *diceSpots*$_2$ are numbers between 1 and 6, but the game is already over, the line has:

  `The game is over`↩

Exit-Command

The exit-command indicates that we want to terminate the execution of the program. The line with the exit-command has:

`exit`↩

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 form, where *winnerColour* stands for the colour of the player who won the game:

  winnerColour `won the game!`↩

Invalid Commands

Whenever the user writes a line that does not start with the words “player”, “square”, “status”, “dice” or “exit”, the game state must not be changed and the program must write a line with:

`Invalid command`↩

4 Examples

Some examples are presented. The left column illustrates the interaction: the input is written in blue and the output in black. All input and output lines end with a newline character, which is omitted for the sake of readability. The right column has information for the reader, serving only to remind you of the rules described above. In this column, "**S**" abbreviates "the player stops on square" and "**P**" abbreviates "his penalty is".

Example 1

`RGB`	Player colours: R, G and B (who play in this order).
`25`	The board has 25 squares.
`2`	2 squares have a fine.
`7 17`	Fines at squares 7 and 17.
`3`	3 squares have a cliff.
`6 20 24`	Cliffs at squares 6, 20 and 24.
`player`	
`Next to play: R`	
`square B`	
`B is on square 1`	
`dice 1 6`	R rolls the dice; **S** 8; **P** 0.
`dice 6 3`	G rolls the dice; **S** 10; **P** 0.
`player`	
`Next to play: B`	
`square G`	
`G is on square 10`	
`status R`	
`R can roll the dice`	
`dice 3 6`	B rolls the dice; **S** 10; **P** 0.
`dice 1 1`	R rolls the dice; **S** 10; **P** 0.
`dice 6 6`	G rolls the dice; **S** 22; **P** 0.
`dice 4 5`	B rolls the dice; **S** 19; **P** 0.
`dice 4 3`	R rolls the dice; **S** 17; **P** 2. Square 17 has a fine.
`square G`	
`G is on square 22`	
`square R`	
`R is on square 17`	
`square B`	
`B is on square 19`	
`status B`	
`B can roll the dice`	
`status R`	
`R cannot roll the dice`	
`dice 5 5`	G rolls the dice; **S** 25; **P** 0. The game ends.
`status B`	
`The game is over`	
`square G`	
`G is on square 25`	
`exit`	
`G won the game!`	

Example 2

`WYM`	Player colours: W, Y and M (who play in this order).
`25`	The board has 25 squares.
`2`	2 squares have a fine.
`7 17`	Fines at squares 7 and 17.
`3`	3 squares have a cliff.
`6 20 24`	Cliffs at squares 6, 20 and 24.
`square R`	
`Nonexistent player`	
`dice 2 4`	W rolls the dice; **S** 7; **P** 2. Square 7 has a fine.
`dice 5 3`	Y rolls the dice; **S** 18; **P** 0. Square 9 has a goose.
`eXiT`	
`Invalid command`	
`dice 1 1`	M rolls the dice; **S** 3; **P** 0. W plays; **P** 1.
`casa red`	
`Invalid command`	
`player`	
`Next to play: Y`	
`status W`	
`W cannot roll the dice`	
`dice 1 1`	Y rolls the dice; **S** 16; **P** 0. Square 20 has a cliff.
`dice 1 7`	
`Invalid dice`	
`dice 2 1`	M rolls the dice; **S** 1; **P** 0. Square 6 has a cliff. W plays; **P** 0.
`square W`	
`W is on square 7`	
`player`	
`Next to play: Y`	
`status W`	
`W can roll the dice`	W will roll the dice when it is his turn to play again.
`dice 4 4`	Y rolls the dice; **S** 8; **P** 0. Square 24 has a cliff.
`dice 2 3`	M rolls the dice; **S** 1; **P** 0. Square 6 has a cliff.
`square Y`	
`Y is on square 8`	
`square M`	
`M is on square 1`	
`dice 5 6`	W rolls the dice; **S** 25; **P** 0. Square 18 has a goose. The game ends.
`dice 0 4`	
`Invalid dice`	
`dice 1 4`	
`The game is over`	
`square Y`	
`Y is on square 8`	
`status Y`	
`The game is over`	
`status marca gira!`	
`Nonexistent player`	
`exit`	
`W won the game!`	

Example 3

`XYZ`	Player colours: X, Y and Z (who play in this order).
`18`	The board has 18 squares.
`2`	2 squares have a fine.
`5 10`	Fines at squares 5 and 10.
`1`	1 square has a cliff.
`11`	Cliff at square 11.
`dice 2 2`	X rolls the dice; **S** 5; **P** 2. Square 5 has a fine.
`dice 2 3`	Y rolls the dice; **S** 6; **P** 0.
`dice 6 3`	Z rolls the dice; **S** 10; **P** 2. Square 10 has a fine. X plays; **P** 1.
`status X`	
`X cannot roll the dice`	
`status Z`	
`Z cannot roll the dice`	
`player`	
`Next to play: Y`	
`dice 1 1`	Y rolls the dice; **S** 8; **P** 0. Z plays; **P** 1. X plays; **P** 0.
`status X`	
`X can roll the dice`	X will roll the dice when it is his turn to play again.
`status Z`	
`Z cannot roll the dice`	
`player`	
`Next to play: Y`	
`dice 2 1`	Y rolls the dice; **S** 5; **P** 0. Square 11 has a cliff. Z plays; **P** 0.
`status X`	
`X can roll the dice`	
`status Y`	
`Y can roll the dice`	
`status Z`	
`Z can roll the dice`	
`player`	
`Next to play: X`	
`dice 3 2`	X rolls the dice; **S** 10; **P** 2. Square 10 has a fine.
`square X`	
`X is on square 10`	
`status X`	
`X cannot roll the dice`	
`status Y`	
`Y can roll the dice`	
`status Z`	
`Z can roll the dice`	
`square Y`	
`Y is on square 5`	
`square Z`	
`Z is on square 10`	
`player`	
`Next to play: Y`	
`exit`	
`The game was not over yet...`	

5 Programming Project Submission

The programming project is submitted to Mooshak. You must submit a `.zip` file to Problem A of contest **IP2223-P1**. Do not forget that:

- The archive should only contain 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 8.[1]

Each student must register for the IP2223-P1 contest according to the following rules:

- The **username** (in Mooshak) must be the **student's number**.
- The **group** (in Mooshak) corresponds to the lab classes shift.
- The **email** address (to which Mooshak sends the password) must be the institutional address.

For example, the student with number 98765, enrolled in shift P3, is the user with name 98765 and belongs to group P3. Only programs submitted to the IP2223-P1 contest by users who respect these rules will be evaluated.

The IP2223-P1 contest opens on the 27th of October and ends at **20:00** on the **7th of November 2022** (Monday). 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 Professor Artur Miguel Dias (amd@fct.unl.pt) up to one hour after the contest closes, indicating the number of the submission that you want to be evaluated.

6 Assessment Criteria

According to the Regulamento de Avaliação de Conhecimentos da FCT NOVA (FCT NOVA Knowledge Assessment 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 assessment, unauthorized information or equipment;
  (b) You give or receive unauthorized help in exams, tests, or any other component of the knowledge assessment;
  (c) You give or receive help, not permitted by the rules applicable to each case, in carrying out of practical assignments, reports or other assessment elements.
- Students directly involved in a fraud are immediately excluded from the course assessment process, without prejudice of a disciplinary or civil procedure.

[1] This information is irrelevant if you only use the subset of the Java language taught in classes because, in that case, the program should have the same behaviour on your machine and on Mooshak.

The document Colaboração Permitida e Não Permitida, available in Moodle, clarifies the points transcribed above.

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** (out of 20)

  A program submitted to the contest that only uses the allowed library classes and that gets P points (out of 120) in Mooshak will be scored P/10 points (out of 20).[2]

  The allowed library classes are **only those that were used in theoretical-practical classes.**

- **Code quality: 8 points** (out of 20)[3]

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

  - **Several classes that characterize 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);
  - Preconditions in public methods;
  - Correct indentation (remember Eclipse's CTRL-SHIFT-F command), lines with 100 characters (maximum)[4] and methods with 25 lines (maximum);
  - A comment at the beginning of each class, which indicates what the objects of the class represent, and a comment before each method, which briefly explains what the method does.

Good luck!

[2]The program can be done incrementally. For example, you can start by assuming that no player stops on a fine, a cliff, or a goose. Of course, as long as the program does not produce the correct results for all tests in Mooshak, it will not get 120 points.

[3]Notice that the quality of the code has a great weight in the grade of the programming project.

[4]When counting the number of characters in a line of code, consider that a tab is equivalent to 4 characters.