Third Practical Exercise

Programming

Grado en Ingeniería Informática

Universidad Carlos III de Madrid

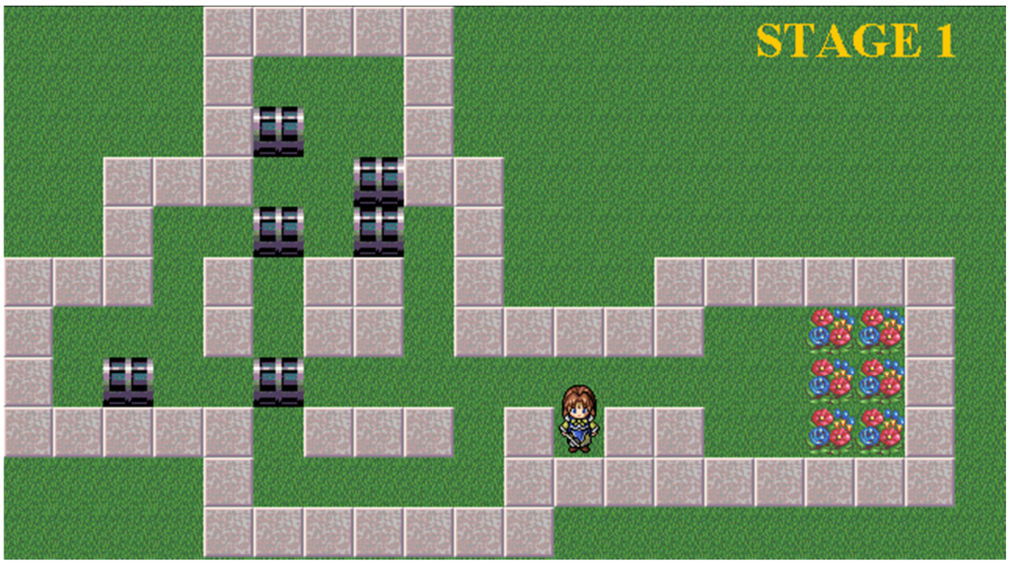
Programming – Grado en Ingeniería Informática	
Authors *Of the English version:* Juan Gómez Romero *Based on the work by:* Ángel García Olaya Manuel Pereira González Gustavo Fernández-Baillo Cañas	

Third Practical Exercise: Sokoban Game

In this exercise, students will develop a Java program to play Sokoban. In this game, the player pushes boxes around in the board, trying to get them to storage locations. The game proceeds as follows:

- At the beginning of the game, the player character is located on the initial position. The player can move the character by using the keyboard
- The character can either move onto empty board cells or push a box
- To push a box, the cell where the box is moved to must be an empty cell or a storage cell
- The game ends when all the boxes are on storage cells, or the player gives up

More information about Sokoban can be found on the Web [play] [Wikipedia].



1. First part: Methods

The first part of the practical exercise is focused on exercises with methods. A class named *FirstPart*, including the methods described below, must be developed. During the development of *FirstPart*, the *main* method will include calls to the methods to test them. At the end of the development process, the code in the *main* method will be changed to allow playing the Sokoban game.

A 2-dimensional array of characters will be used to represent the state of the board, according to the following correspondence:

- 'H' (uppercase 'h'): Walls
- ' ' (space): Empty cell
- 'O' (uppercase 'o'): Storage cell
- '*' (asterisk): Box on an empty cell
- 'X' (uppercase 'x'): Stored box / Box on a storage cell
- 'a' (lowercase 'a'): Player character on an empty cell
- '@' ('at' symbol): Player character on a storage cell

For example, this array represents a valid Sokoban game board (11x19):

```
    HHHHH
    H   H
    H*  H
  HHH  *HH
  H  * * H
HHH H HH H   HHHHHH
H   H HH HHHHH  OOH
H *  *         OOH
HHHHH HHH HaHH  OOH
    H     HHHHHHHHH
    HHHHHHH
```

A simpler board is the following (8x5):

```
 HHH
 HOH
 H H
 H*H
HH HH
H   H
HHaHH
 HHH
```

At the beginning of the game, the board is printed on the screen. Next, the user is asked to enter a move command (keys A, W, D, X to move; Q to end). The move is checked and, if valid, it is performed, thus updating the board; otherwise, a corresponding message is displayed.

This is the output of a Sokoban game:

```
 HHH
 HOH
 H H
 H*H
HH HH
H   H
HHaHH
 HHH

move? w

 HHH
 HOH
 H H
 H*H
HH HH
H a H
HH HH
 HHH

move? w

 HHH
 HOH
 H H
 H*H
HHaHH
H   H
HH HH
 HHH

move? w
```

```
 HHH
 HOH
 H*H
 HaH
HH HH
H   H
HH HH
 HHH

move? w
Good! You completed the game!

 HHH
 HXH
 HaH
 H H
HH HH
H   H
HH HH
 HHH
```

1.1 Tasks

The implementation with methods of Sokoban must include two classes, *Position* and *FirstPart*, which will be inside the package *programming.partone*.

Position is a data structure to represent the position of a cell in the board. This class has two attributes:

- `int row`: integer number that stores the row of the cell
- `int col`: integer number that stores the column of the cell

For example, given the simple board above, the initial position of the player character is row: 6, col: 2.

FirstPart includes the following **STATIC** methods to play Sokoban:

- *main*: This is the starting point of the program. At the beginning of the implementation, the *main* method is empty. As long as additional methods are implemented, students must include calls to them in the *main*, in order to test if they are correct. At the end of the implementation, the *main* will only include a call to the *play* method
- *generateBoard*: Creates a 2-dimensional array of characters storing the initial state of the game
  - Parameters
    - None
  - Returns
    - `char [][]`: Initial board
  - Action
    - Allocates and initializes an array with the initial representation of the board. Returns this array
    - The initial board is the 8x5 simple board
- *printBoard*: Prints the array representing the board
  - Parameters
    - `char [][] board`: Board to print
  - Returns

        - Nothing
    - Action
        - Prints on the screen the array `board`

- *getUserInput*: Reads next move
    - Parameters
        - None
    - Returns
        - Read character
    - Action
        - Prints on the screen a message for the user and reads the input from the keyboard. Returns a single character, which must be 'W', 'A', 'X', 'D' or 'Q'

- *findPlayerPosition*: Finds player character position
    - Parameters
        - `char [][] board`: Board to search in
    - Returns
        - `Position`: Position of the player character on the board. If player is not found, returns *null*
    - Action
        - Searches the player character on the board ('a' or '@'). Returns the position where the character was found

- *playerWins*: Tests if all the storage positions are filled with boxes
    - Parameters
        - `char [][] board`: Board to test
    - Returns
        - `boolean`: true, if there are no cells with value 'O' (storage) or '@' (player on storage); otherwise, false
    - Action
        - Searches cells with values 'O' or '@'. If a cell with any of these values is found, the method ends and returns false; otherwise, the method returns true

- *getPositionAfterMove*: Calculates the position which would result after making a move
    - Parameters
        - `Position currentPos`: Original position
        - `char direction`: direction to move to ('W', 'A', 'D', 'X')
    - Returns
        - `Position`: Position after move. If move is not valid, the returned position is *null*
    - Action
        - Creates a new `Position` object representing the position that would result after moving `currentPos` to `direction`

- *makeMove*: Performs a move on the board
    - Parameters

        - `char[][] board`: Game board
        - `char direction`: direction to move to ('W', 'A', 'D', 'X')
    - Returns
        - `boolean`: true if the move was successful; otherwise, false
    - Action
        - First, the method finds the current position of the player character (use *findPlayerPosition)*
        - The method obtains the position that would result after moving the player character in `direction` (use *getPositionAfterMove)*
        - If the position is inside the board, the method must test if the movement is valid according to the current state of the board. If so, the board will be conveniently updated.

          Please notice that several situations must be considered: the player moves from an empty cell to an empty cell, the player moves from a storage cell to an empty cell, the player moves a box to an empty cell, the player moves a box to a storage cell, etc.

- *play*: Plays Sokoban game
    - Parameters
        - None
    - Returns
        - Nothing
    - Action
        - Implements the game loop: while the game continues, the program
            1. Generates a new board
            2. Prints the board
            3. Gets player input
            4. Makes move
            5. Checks if player won
                a. True: game ends
                b. False: game continues (step 2)

1.2 Optional tasks

Optional tasks allow students to obtain extra points –please notice that the grade in this exercise is 10 at most. In this part, two optional tasks are proposed:

1. Display the number of "steps" done by the player and the number of stored boxes when the board is printed
2. Support several stages. When a scenario is solved, the player can continue with the next stage (Implement 4-5 stages)

2. Second part: Classes, objects and GUI

In the second part of the exercise, students will implement again the Sokoban game by relying on classes and objects. The board will be shown now in graphical mode. Previous methods can be partially reused.

Students are provided with two initial files: *Sokoban.java* and *MySokoban.java*:

- *Sokoban*: Includes several functionalities to deal with the graphical interface. **DO NOT MODIFY** this class
- *MySokoban*: Students will complete this class will be completed to accomplish the requirements of the exercise

These files are included inside the package *programming.parttwo*. They can be downloaded from Aula Global.

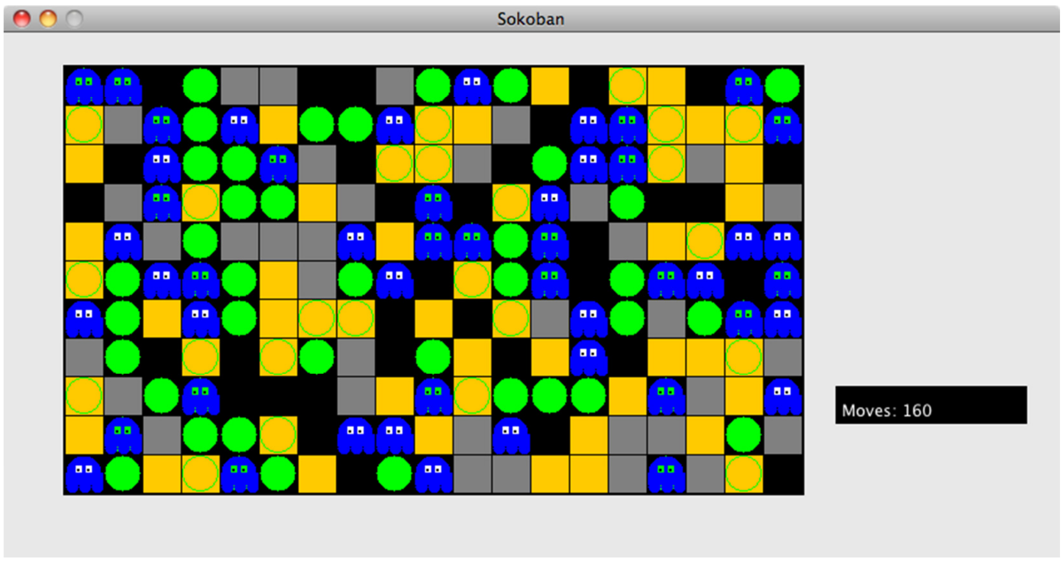
Students must create two additional classes:

- *Position*: Class to represent a position of the board
- *Board*: Class to represent a game board

2.1 Starting tasks

Before starting with the solution of the second part, students are recommended to study the code of the class *MySokoban* to understand how it works. To do so, please follow the indications below:

- Run *MySokoban*. The initial implementation of this class just prints on the screen random cell values. If any key is pressed, the cells are printed again with new random values and the step counter is incremented in 10. 'Q' can be used to end the program.



- Study *MySokoban*. Look carefully the code of *MySokoban*:
    - `enum Cell`: This enumerated type is used to represent allowed cell values (The new implementation does not use an array of characters to represent the state of the board, but an array of `MySokoban.Cell`)
    - Constants `ROWS`, `COLS`: Size of the board
    - `public void processKey(char key)`: This method is automatically called when a key is pressed

    - `public Cell getCellFromBoard(int row, int col)`: This method is used to retrieve a cell in a given position. In the initial implementation, no board is created and random cells are returned
    - `public int getMoves(), public void setMoves(int moves)`: Get/set methods for private attribute `moves`
- Test *MySokoban*. Solve these preliminary tasks:
    - Remove the method *getRandomCell*. Change the method *getCellFromBoard* to return a *WALL* cell at any case
    - Modify the method *processKey* to increment the step counter in a random value in {1, …, 100} after any key pressing
    - Change the board size
    - Extend the method *processKey* to increment the step counter in 1000 after pressing key 'P'

2.2 Tasks

The new implementation of Sokoban must include four classes, which will be inside the package *programming.parttwo*.

Sokoban **MUST NOT BE MODIFIED** by the students.

Position is a class to represent the position of a cell in the board. The contents of this class are:

- Attributes (private)
    - `int row`: integer number that stores the row of the cell
    - `int col`: integer number that stores the column of the cell
- Constructor
    - *Position*
        - Parameters
            - `int row`: Initial row value
            - `int col`: Initial column value
        - Action
            - Initializes attributes with the values passed as parameters
- Methods (non static)
    - *getRow*, *getCol*: get methods for private attributes
        - Parameters
            - None
        - Returns
            - Attribute value
        - Action
            - Returns the current value of a private attribute
    - *getPositionAfterMove*: Calculates the position which would result after making a move
        - Parameters
            - `char direction`: direction to move to ('W', 'A', 'D', 'X')
        - Returns
            - `Position`: Position after move. If move is not valid, the returned position is *null*
        - Action

- Creates a new `Position` object representing the position that would result after moving this position in `direction`

Board is a class to represent the cells of the game. The contents of this class are:

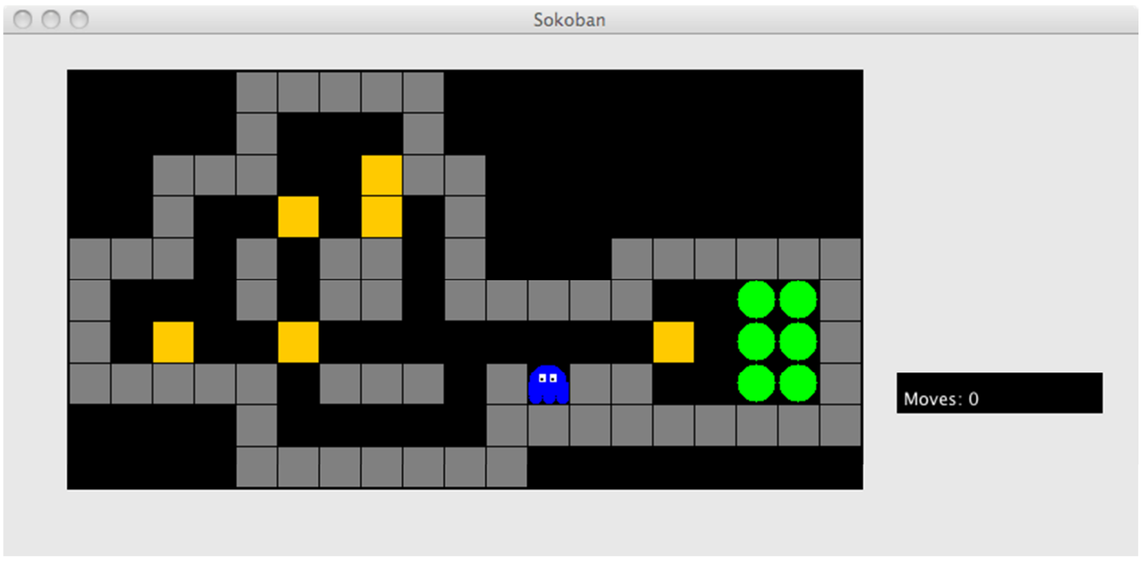
- Attributes (private)
    - `MySokoban.Cell [][] b`: board array
- Constructor
    - *Board*
        - Parameters
            - `int rows`: Number of rows
            - `int cols`: Number of columns
        - Action
            - Initialize the attribute b according to the configuration in the complex 11x19 board
- Methods (non static)
    - *getCell*: get cell value at specified position
        - Parameters
            - `int row`: Row
            - `int col`: Column
        - Returns
            - `MySokoban.Cell`: Board value
        - Action
            - Returns the board value at position (`row`, `col`)
    - *findPlayerPosition*: Finds player character position
        - Parameters
            - None
        - Returns
            - `Position`: Position of the player character on the board. If player is not found, returns *null*
        - Action
            - Searches the player character on the board (cells `MySokoban.Cell.PLAYER` or `MySokoban.Cell.PLAYER_AT_STORAGE`). Returns the position where the character was found
    - *playerWins*: Tests if all the storage positions are filled with boxes
        - Parameters
            - None
        - Returns
            - `boolean`: true, if there are no cells with value `MySokoban.Cell.STORAGE` or `MySokoban.Cell.PLAYER_AT_STORAGE`; otherwise, false
        - Action
            - Searches cells with values `STORAGE` or `PLAYER_AT_STORAGE`. If a cell with any of these values is found, the method ends and returns false; otherwise, the method returns true
    - *makeMove*: Performs a move on the board

- Parameters
  - `char direction`: direction to move to ('W', 'A', 'D', 'X')
- Returns
  - `boolean`: true if the move was successful; otherwise, false
- Action
  - First, the method finds the current position of the player character (use *findPlayerPosition* in `Board`)
  - The method obtains the position that would result after moving the player character to `direction` (use *getPositionAfterMove* in `Position`)
  - If the position is inside the board, the method must test if the movement is valid according to the current state of the board. If so, the board will be conveniently updated.

    Please notice that several situations must be considered: the player moves from an empty cell to an empty cell, the player moves from a storage cell to an empty cell, the player moves a box to an empty cell, the player moves a box to a storage cell, etc.

MySokoban must be extended to play the game.

- Add an attribute with type `Board` named `theBoard`
- Extend the constructor of `MySokoban` to allocate memory for the attribute `theBoard`
- Modify the method *getCellFromBoard* to retrieve the cell at position (`row`, `col`) of `theBoard`
- Modify the method *processKey* to:
  1. Make move according to pressed key
  2. If move was successful
     a. Update moves counter
     b. Test if player won. If player won, end the game



2.3 Optional tasks

Optional tasks allow students to obtain extra points –please notice that the grade in this exercise is 10 at most. In this part, two optional tasks are proposed:

1. Display the number of stored boxes
2. Support several stages. When a scenario is solved, the player can continue with the next stage (Implement 4-5 stages) (All stages can have the same size)

3. Evaluation

The exercise will be graded according to the following criteria:

- **TASK ACHIEVEMENT (6)**
  - **First Part (3)**
    - *Position class*: 0,2
    - *main*: 0,1
    - *generateBoard*: 0,1
    - *printBoard*: 0,1
    - *getUserInput*: 0,2
    - *findPlayerPosition:* 0,3
    - *playerWins*: 0,3
    - *getPositionAfterMove*: 0,4
    - *makeMove*: 0,9
    - *play*: 0,4
  - **Second Part (3)**
    - Class ***Position*** **(0,5)**
      - Attributes, get/set: 0,1
      - *Constructor*: 0,1
      - *getPositionAfterMove*: 0,3
    - Class ***Board*** **(1,5)**
      - Attributes: 0,1
      - *getCell*: 0,3
      - *findPlayerPosition*: 0,2
      - *playerWins*: 0,3
      - *makeMove*: 0,6
    - Class ***MySokoban*** **(1)**
      - Attributes: 0,1
      - *main*: 0,3
      - *getCellFromBoard*: 0,2
      - *processKey*: 0,4
- **REPORT (2)**
  - Presentation, exposition, ortography: 0,5
  - Quality of technical and user manual: 1
  - Justification of design decisions: 0,5
- **CODE (2)**
  - Quality of the implementation (efficiency, user input and error management): 1
  - Quality of code and comments: 1
- **EXTENSIONS (1)**
  - Section 1, display number of steps and stored boxes: 0,2
  - Section 1, multiple stages: 0,3
  - Section 2, display number of stored boxes: 0,1
  - Section 2, multiple stages: 0,4
  - Other improvements will be also considered