

Object Oriented Programming - Exercise 1: Flow Control and Working with Objects

Contents

1 Objectives	1
2 The Nim game	2
2.1 Game rules	2
2.2 Move examples	2
3 Exercise definition	3
3.1 Program arguments	3
4 The classes composing the program	4
4.1 The Competition class	4
4.2 The Board class	4
4.3 The Player class	5
4.4 The Move class	7
5 Working with the provided classes	8
5.1 Importing the provided classes into your workspace	8
5.2 Importing external Java classes	8
5.3 Using classes in your code	8
6 Nim Competition program Flow	9
7 Additional guidelines	11
8 Suggested guidelines	11
9 School solution	12
10 Grading	12
11 Testing and submission	12

1 Objectives

This exercise's purpose is to get you comfortable both with basic programming in Java and basic programming in the Eclipse IDE. It focuses on the flow principles taught in class (such as loops and conditions) and on creating classes. You will both use provided classes (supplied by us) and

write additional classes on your own. Hopefully you would get the notion of how intuitive and natural it is to integrate objects into your code (code examples are provided).

This description document is long **not** because the exercise itself is particularly long or hard, but for the purpose of guiding you, save you time and make the task as clear as possible. **So read the entire document first!**

2 The Nim game

The [Nim game](#) is a mathematical game of strategy for two players. While many variations of the game exist, you will implement a very specific version of the game that will be referred to as '*the Nim game*' throughout this document.

2.1 Game rules

The board of the game consists of 16 sticks, ordered in 4 rows of 7,5,3 and 1 sticks, respectively.

Here is a sketch of the board:

```
{Row no. 4 }      I
{Row no. 3 }      I I I
{Row no. 2 }      I I I I I
{Row no. 1 }      I I I I I I I
```

*The indexing convention here would be to start counting from 1 (as opposed to counting from 0). That means that the first row is **row no. 1**, and the 1st and leftmost stick in some row is **stick no. 1**.

The two players take alternating turns, where in each turn a player **must** mark exactly one continuous sequence of sticks from one (and only one) of the rows; For example: the sequence starting at the 1st stick of the 3rd row and ending at the 3rd (and last) stick of the row, or the sequence starting at the 2nd stick of the 1st row and ending at the 5th stick of the same row, or even the sequence starting and ending at the 1st stick of the 2nd row. The player that marks the last stick - meaning that after his move all sticks on the board are marked - is the **loser**; so the goal of each player is to mark sticks in a way that will force the other player to mark the last stick.

After a move is made and the respective sticks are marked; future moves cannot include or skip over marked sticks.

2.2 Move examples

For example, if the first player marked the 3 middle sticks of the first row (row=1, left bound=3, right bound=5), it leads to the following board state (0s denotes marked sticks):

```
{Row no. 4 }      I
{Row no. 3 }      I I I
{Row no. 2 }      I I I I I
{Row no. 1 }      I I 0 0 0 I I
```

Now there are only 6 legal moves on the 1st row: (1) Marking the 1st stick. (2) Marking the 2nd stick. (3) Marking the sequence starting at the 1st stick and ending at the 2nd stick. (4) Marking the 6th stick. (5) Marking the 7th stick. (6) Marking the sequence starting at the 6th stick and ending at the 7th stick.

Some **illegal** moves on the 1st row are: (1) Marking the 3rd stick. (2) Marking the sequence starting at the 1st stick and ending at the 7th stick. (3) Marking the sequence starting at the 1st stick and ending at the 3rd stick. (4) Marking the sequence starting at the 3rd stick and ending at the 5th stick.

Other illegal moves: (5) In the 3rd row, marking the sequence starting at the 1st stick and ending at the 4th stick (only three sticks in the row). (6) Any move on the 5th, 8th or the 0th row (no such rows).

3 Exercise definition

Your task in this exercise is to write a Java program that runs a Nim game competition, composed of several rounds, between two players. Each player can be either human - who must then interact with the program - or controlled by a computer agent - in which case it can be one of several types of computer agents. Specifically, you would have to implement the logic of the game, as described below, across four different Java classes.

The program you will write should:

1. Conduct a Nim competition of an arbitrary number of rounds between two players of any type (two humans, two randoms, human vs smart, etc.).
2. Interact with any human players according to a certain protocol, which includes displaying the board and receiving their moves as input.
3. Keep track of the number of victories of each competing player, and display them when the competition ends.
4. Run efficiently. We do not demand state-of-the-art efficiency; just make sure you can run 1,000 rounds of two of both your 'smart' and random agents in under 5 seconds (this should not be hard to achieve).

3.1 Program arguments

The game you will implement has three user-specified parameters:

1. The type of the first player, which is a positive integer between 1 and 4: 1 for a Random computer player, 2 for a Heuristic computer player, 3 for a Smart computer player and 4 for a human player.
2. The type of the second player, which is likewise a positive integer between 1 and 4.
3. The number of rounds to be played in the competition.

The **Competition** class will be the main class of the program, and so it will be the only class with a **main** method, and the program arguments will be passed to this method. For example, once you have all required *.java files compiled into *.class files, including a **Competition.class**

file, to get a competition of 58 rounds between a human player and a random player, you should run the program in the shell console using:

```
java Competition 4 1 58
```

*To run the program with arguments using Eclipse: right click on Competition.java→Run As→Run Configurations. In the Arguments tab, in Program arguments, type: 4 1 58

4 The classes composing the program

The four classes composing the program, including the two classes you need to implement from scratch, are well documented in a [publicly-accessible Javadoc](#); Javadoc is a collection of HTML files (so you can read them with your internet browser) that display, for each class, the list of public fields and methods it contains, and for each such method details the return type and the parameters it receives (this is called the method's signature).

You must implement all the public methods indicated in the Javadoc with exactly the same signature as in the documentation. You are not allowed to add any other **public** methods or fields to your code (but you can add as many private methods and fields as you want).

The descriptions below only describe in general the different methods. In the Javadoc you will find exact specifications for the behavior of the different methods, according to which you are required to implement the methods.

4.1 The Competition class

Defined by the `Competition.java` file. You are supplied with a skeleton of this class, which only includes the implementation of handling program arguments. An instance of this class represents a Nim competition between two players, consisting of a given number of rounds. The Competition object also keeps track of the number of victories of each player. The public methods of this class are (return types are written in **purple**):

- **public** `Competition(Player player1, Player player2)` - The class constructor, which receives two Player objects representing the two competing opponents.
- **public int** `getPlayerScore(int playerPosition)` - Returns the number of victories of a player. `playerPosition` should be 1 or 2, corresponding to the first or the second player in the competition.
- **public void** `playMultiple(int numRounds)` - Runs the competition for "numRounds" rounds.
- **public static void** `main(String[] args)` - The main method. Should create a Competition object according to the given arguments, run the competition and print any required messages, including the results of the competition. You are not required to check the validity of the arguments.

4.2 The Board class

Defined by the `Board.java` file. You are given the full implementation of the Board class. Presented here is the API of the class; you can (and should) use all the public methods of this class,

according to their description, in your code. **You may not change the code in this file in any way.**

The Board class represents a board of the Nim game. The board can be of a variable size. In this implementation it only has 4 rows, 7 sticks in the first row, and 5,3 and 1 sticks in the following rows. A board object is disposable, meaning that the moves performed on the board are not reversible, and if a "clean" board is required, the user has to instantiate a new board object. The public methods of this class are (return types are written in **purple**):

- **public** Board() - The class constructor, which initializes a clear board.
- **public int** getNumberOfMarkedSticks() - Returns the number of marked sticks in the current state of the board.
- **public int** getNumberOfUnmarkedSticks() - Returns the number of **unmarked sticks (i.e in its original state)** in the current state of the board. Note that when this method returns 0, that means that all the sticks in the board are marked.
- **public String** toString() - Returns a multi-line human-readable visual representation of the board as a String object. Can be used for printing the board to screen and for debugging.
- **public int** getNumberOfRows() - Returns the number of rows in the board
- **public int** getRowLength(int row) - Returns the number of sticks in the row.
- **public boolean** isStickUnmarked(int row,int stickNum) - Given an index to the stick position (row and number in row - counting from the left side), this method returns true if the stick is unmarked, and returns false in case that the stick is marked or the index is out of bounds.
- **public int** markStickSequence(Move move) - Makes an attempt to mark the given sequence of sticks. In case the move is not legal the board is not updated. There are two types of illegal moves: the move exceed the board bounds (-1 is returned), and the move crossed marked sticks (0 is returned). If the move is legal, the board is updated and 1 is returned.

4.3 The Player class

Defined by the Player.java file. You are given a partial implementation of this class.

The player class interacts with the Board class in order to reason about the game. Each player is initialized with a type, either human or one of several computer strategies, which defines with which move he responds when given a board in some state.

In the class Player, there are 4 constant values that define the mapping between the player type and the code of that type. For instance, code mapping for the random player is defined by:

```
public static final int RANDOM = 1;
```

Which means that type 1 stands for the random player. Since it is a static variable, you can access it from every class in your program as: **Player.RANDOM**. See 5.3 for an example how to use it.

The public methods of this class are:

- `public Player(int type, int id, Scanner scanner)` - The class constructor, which initializes a new player. The scanner is used only if the player is human, but we need to provide it for every type. Further explanation in subsection 5.3
- `public int getPlayerType()` - Returns an integer matching the player type.
- `public int getPlayerId()` - Returns the player's id.
- `public String getPlayerTypeString()` - Returns a String matching the player type.
- `public Move produceMove(Board board)` - This method receives a Board object representing the current state of the game, and returns a Move object representing the move the player wishes to perform in this state. Thus, it encapsulates all the reasoning of the player about the game.

The choice of the move depends on the type of the player: a human player chooses his move manually; the random player should return some random move; the Smart player can represent any reasonable strategy; the Heuristic player uses a specific strong heuristic to choose a move.

In practice this method calls another method depending on the type of the player (see the provided `Player.java` file). You will need to implement the player specific methods according to the following descriptions:

The Heuristic player

A Heuristic is a strategy that is usually based on intuition, works well, and there's no proof why. It's very hard to beat the heuristic player, you can try and see for yourself. You receive a complete implementation of the Heuristic strategy, so the corresponding method is already implemented.

The Human player

For the Human player type, you must interact with the user to obtain the move. To receive integers as input, use the `nextInt()` method of the `Scanner` object to read that integer input, for example:

```
int userInput = scan.nextInt();
```

*More about Scanner, in section 5.3

The interaction should be according to the following protocol:

1. Print: "Press 1 to display the board. Press 2 to make a move:"
2. Receive input from the user.
 - (a) If 1 is typed then the board is printed (as explained in subsection 5.3), and go back to step 1.
 - (b) If 2 is typed, ask for details:
 - Print: "Enter the row number:" and receive input
 - Print: "Enter the index of the leftmost stick:" and receive input

- Print: "Enter the index of the rightmost stick:" and receive input
- (c) For any other input, print: "Unknown input." and return to step 1

*The human player does not need to check if the input values for the move are valid.

* You need to make sure that the messages you print are exactly the same as in the school solution.

The Random player

For the Random player type, you should implement some random way to choose moves. **The random player must return a valid move (see sections 2 and 4.2).** For a given board state, there can be many valid moves. Think of a way you can randomly choose one of them and return it. It does not have to be totally random, and you don't need to choose each move with equal probability, you just need to make sure it is not deterministic (i.e. it returns different moves for the same board state when called several times).

Random integers can be easily generated in Java by first instantiating a Random object with `Random myRandom = new Random();` and then calling the method `nextInt(int n)`. For example, to get a number between 1 and 5, you can call `int randomInt = myRandom.nextInt(5) + 1;`

The Smart player

For the Smart player type, implement the best strategy you can think of. The smart player is at least expected to perform better than the random player (Notice that it is easy to beat the random player. For example, think of the situation when there are sticks left in only one row...). The smart player also must return a valid move. **Explain you implementation of the Smart player in the README file!**

4.4 The Move class

Defined by the `Move.java` file. You are not supplied with an implementation of this class, and so you need to write it from scratch! The Move class represents a move in the Nim game by a player. A move consists of the row on which it is applied, the left bound (inclusive) of the sequence of stick to mark, and the right bound (inclusive) of the same sequence. The public methods of this class are:

- `public Move(int inRow, int inLeft, int inRight)` - The class constructor, which receives the parameters defining the move.
- `public String toString()` - Returns a string representation of the move. For example, if the row for the move is 2, the left bound is 3 and the right bound is 5, this function will return the string "2:3-5" (without any spaces).
- `public int getRow()` - Returns the row on which the move is performed.
- `public int getLeftBound()` - Returns the left bound of the stick sequence to mark.
- `public int getRightBound()` - Returns right bound of the stick sequence to mark.

5 Working with the provided classes

5.1 Importing the provided classes into your workspace

To be able to use the given classes you should perform a few short steps that will import the supplied code into your project. Using the Eclipse IDE, follow these steps:

- Download `nim.jar` from the course moodle page, and extract the supplied files from the archive.
- Right click the (default package) of your project in the "Package Explorer" view.
- Click the `Import...` context-menu option.
- Under the `General` category choose `File System` and click `Next`.
- Click the `Browse...` button and navigate to the location of the extracted files.
- In the right-hand box choose the files `Board.java`, `Competition.java` and `Player.java`.
- Click finish. The source files of the three supplied classes are now in your projects.

5.2 Importing external Java classes

The `Scanner` and `Random` classes are part of the general Java library and have to be imported in order for you to use them. For instance, in every class that uses `Scanner`, you should add at the top of the document (before everything else):

```
import java.util.Scanner;
```

5.3 Using classes in your code

As mentioned before, some of the parts are already implemented and ready for your use: the `Board` and `Player` classes we provided you with. In your implementation of the main method in the `Competition` class you will want to create a `Board` object, probably even several times, which you can do like this:

```
Board board = new Board();
```

Now we can use it for the different tasks required for the implementation of this exercise. For instance, if you want to display a human user with the current state of the board, you can print it to screen using

```
System.out.println(board);
```

This is possible because the `Board` class implements the `toString()` method so that it returns a very human-readable `String` representation of the board, and when the `println()` method receive an object as a parameter it uses the `String` returned by the `toString()` method of that object to print a string representation of it. We will learn more about how this is possible later in the course.

with the correct arguments. You will also create Player objects representing the players competing in the competition. To instantiate a Player object, you will need to supply the constructor with, among other parameters, a Scanner object with which a human Player object can communicate with the user.

Create a scanner object with:

```
Scanner scanner = new Scanner(System.in);
```

The Scanner class allows the user to read values of various types and from various sources. There are far more methods in class Scanner than you will need for this exercise; you will only need to use those that will allow you to read in numeric values from the keyboard without having to convert them from strings and determine if there are more values to be read (see subsection 4.3).

You can now instantiate a Player object with:

```
Player player1 = new Player(Player.RANDOM,1,scanner); //A Player object for a random  
player with id=1
```

Doesn't it feel wrong that we need to instantiate a player object of any type with a scanner, given that only human players will use it? We will soon learn a very natural way to handle problems arising from classes with different functionality for different "sub-types".

In order to know which move a player would like to perform on the board (in a given state), you will have to supply the player with the corresponding Board object:

```
Move player1_move = player1.produceMove(board);
```

Usually when we finish using Scanner, we call the `close()` method of the Scanner object to release the resources the scanner object was using. In this case **we will not do it**, since the resource we were using was actually a resource of the operating system and we are not supposed to close it. Since we don't close the scanner, a compilation warning will be issued. **This is an acceptable warning, all other warnings your program will produce during compilation are unacceptable.**

Don't forget to read the [publicly-accessible Javadoc](#) of the four classes to understand what the methods do and to better understand how they interact.

6 Nim Competition program Flow

The general flow of the Nim competition program is outlined below.

The competition class constructor receives the boolean parameter - "displayMessage", which indicates whether the board should also print messages to the user in the middle of the game (This is called: "verbose mode"). A competition class initialized with `displayMessage=true` should also output the text **marked in red**.

General message outputs, should be always printed (even when `displayMessage` is false), and are **marked in green**.

1. Two players are initialized according to the program arguments. The first is given id=1, the second gets id=2.
2. A new Competition object is created as described in the previous section.
3. The following message is **always** printed:
`Starting a Nim competition of X rounds between a TYPE1 player and a TYPE2 player.`
 Where X is the number of rounds, and TYPE1, TYPE2 are the String representations of the respective player types. Use
`String stringRepOfType = somePlayerObject.getTypeName();`
 to get the correct String representation of the object `somePlayerObject`.
4. Play *N* rounds of the Nim game: (where N is the 3rd argument, described in section 3.1)
 - (a) A new board is initialized.
 - (b) `Print: "Welcome to the sticks game!"`
 - (c) The current player is set to player1 (player1 always starts), meaning that player 1 will take the first turn in the next step:
 - (d) While there are still unmarked sticks on the board:
 - i. `Print: "Player X, it is now your turn!"`
 (X is the player's id)
 - ii. The current Player is "asked" for a move.
 - iii. The board attempts to preform the move (see section 4.2):
 - A. If it is not legal:(see section 2)
 - `Print: "Invalid move. Enter another:"`
 - go back to step 4(d)ii
 - B. If the move is legal:
 - `Print: "Player X made the move: MOVE"`
 (X is the player's id. MOVE is the "toString()" result of the current move)
 - The game board is already updated, nothing else to do.
 - iv. the turn ends, the other player becomes the current Player.
 - (e) The player who has no sticks left to mark is the winner, because the other player marked the last stick. The winner receives one point.
 - (f) `Print: "Player X won!"` (X is the player's id)
5. When the competition ends, the following message is **always** printed:
`The results are wins1:wins2`
 Where `wins1` = the number of victories of player 1, and likewise for `wins2`.

Important note: You should use `displayMessage=true` only when the main method receives as an input at least one human player .

* For testing and debugging purposes at first use `displayMessage=true` for all players. At a later stage, before you submit, send `displayMessage=true` to the Competition constructor only when there is a human player in the game.

You might ask yourself, why not simply let the Competition class check if one of the players is human, and use "verbose mode" only in this case? Well, currently we use verbose for humans only, but what if tomorrow we want to add a new player type and use verbose mode for him as well? The Competition class does not need to depend on the Player types that it is initialized with. It is the programmers responsibility to decide in which scenarios should the Competition class be set up with a "verbose mode".

In any case where there is a conflict between this description and the behavior of the school solution, **the school solution wins!** You are tested against the school solution.

7 Additional guidelines

1. You may add as many private fields and methods as you want to any one of the classes (except for the board). You may not add any additional public methods.
2. Document the code to make it clear (to others and yourself)
3. Your code should be written according to the coding style guidelines (can be found [here](#)).

8 Suggested guidelines

1. Start early!
2. Read the description of the game to understand how it works.
3. Next, run the school solution to see what your program should look like to the user.
4. Acquaint yourself with the provided documentation.
5. Begin by creating a `Move.java` file in you project, and implementing the required public interface.
6. Now import the provided class files (see subsection 5.1): `Board.java`, `Competition.java` and `Player.java`. Since the Heuristic-type logic in `Player` is already implemented, you can already use `Player` objects without adding any more code to the `Player` class, as long as you initialize them to be of the Heuristic type.
7. In the `Competition` class, write the code for a single **turn**: asking the `Player` object for a move once, and updating the board; verify that the state of the board is as expected. You can run your code where both players are of the Heuristic type. Debug.
8. Extend the above code to a full single round. Debug.
9. Extend the above code to a sequence of rounds. Debug.
10. Implement the Random player type (should be the easiest of the three). Debug.
11. Implement the Human player type. Debug.

12. Now that you know more about the game, and you have debugged your way through several play-throughs, try and think of a good strategy of your own, and implement it for the Smart player type. Debug.
13. In order to test how well the Smart player performs, run a competition with many rounds against the random player.
14. Run the provided testers. You guessed it: debug!

9 School solution

An executable version of the school solution can be invoked from the lab computers by typing `~oop/bin/ex1/school` in the shell command line. **Use it** to learn exactly how your program should behave, as the output of your program and the school solution should be **exactly** the same. You can use text comparison tools to check for exact textual match (e.g. the `diff` command in Linux).

10 Grading

The grade for this exercise is based on automatic testing (50%), code review and the answers in your README. In this exercise, you only need to **explain you implementation of the Smart player in the README file**.

90% of the automatic tests will run as soon as you submit your solution. The rest of the tests are hidden and will tackle more subtle aspects of your code. Thus, you are encouraged to test your code on more elaborated scenarios.

11 Testing and submission

- You should submit the following files:
 1. Board.java *(in the same state you got it)*
 2. Player.java *(with a complete implementation)*
 3. Competition.java *(with a complete implementation)*
 4. Move.java *(which you wrote yourself)*
 5. README *(as explained [here](#))*
- Create a JAR file named `ex1.jar` containing only these files by invoking the shell command:
`jar cvf ex1.jar Board.java Player.java Competition.java Move.java README`
The JAR file should not contain any other files.
- The first thing that the testers check is that the jar is OK and contains all required files.
Failing to pass this test will result in automatic 0 grade!

Good Luck !