Java Owari

161 (C1), 176 (J1), Integrated Programming Laboratory

28th November – 5th December 2014

Aims

- To gain further experience in writing loops.
- To practice writing classes and manipulating objects.
- To gain experience in object-oriented design.
- To write a simple game-playing program in Java.

The Problem

Up until now you have only used Java to approach problems in a procedural manner. In this exercise you'll use classes and objects to build an *object-oriented* program. (Note the lectures on Objects will start on Monday 1st December).

The Game of Owari

Owari is a game widely played throughout West Africa. There are no standard rules for Owari and the rules vary between the different places where it is played. The program that you are being asked to write should allow two players to play Owari according to the following rules:

- Owari is played on a board containing 12 bowls divided into two rows of six bowls. At the start of the game each bowl contains 4 stones. In addition, there are two pots used to store stones that have been captured by the players. Initially the pots of captured stones are empty. The starting configuration is shown in Figure 1.
- The two players take it in turn to make moves. Player 1 starts.
- Player 1 can only move stones from the bottom row of bowls, while player 2 can only move stones from the top row.
- A move consists of choosing a bowl, removing all the stones from it and then, moving in an anti-clockwise direction, placing one stone in each bowl following, until all the stones have been placed. Note that the board wraps around with the "last" bowl for one player being next to the "first" bowl for the other player.
- If a player has no stones on their side of the board then they must pass their move. Conversely, a player must move if a move is possible.
- If a bowl contains a single stone before a move is made and ends up with two or more stones after the move, then the player captures the stones in that bowl, moving them to their capture pot.

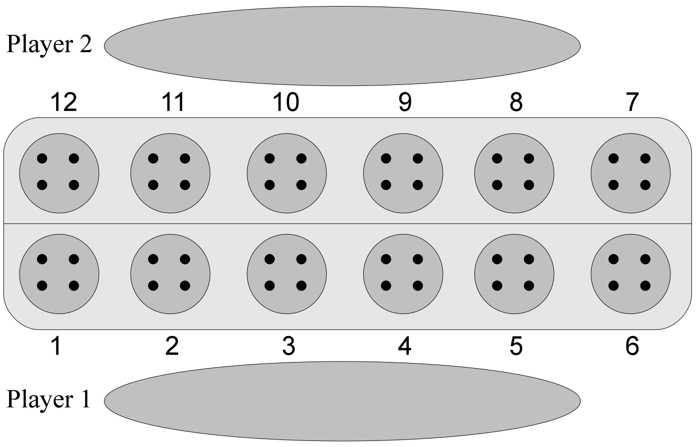

Figure 1: The Owari board at the start of a game.

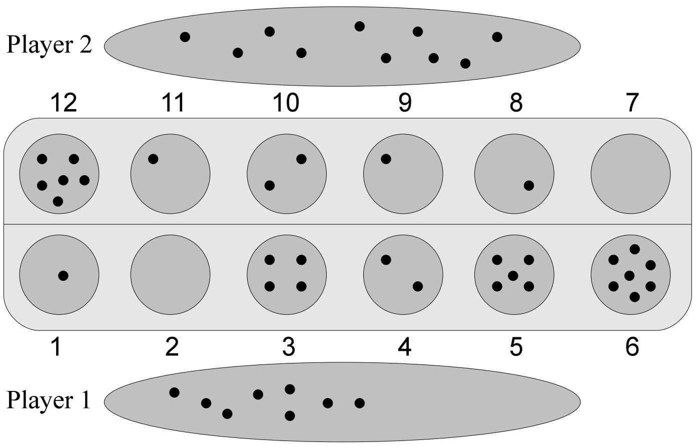

Figure 2: It is player 1's turn to move, and player 1 will choose bowl 6.

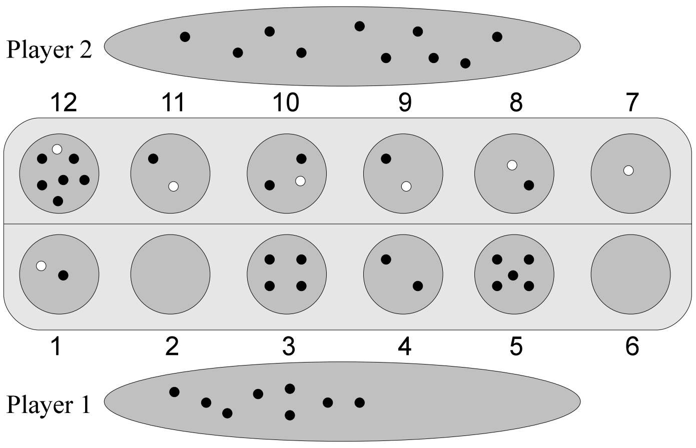

Figure 3: Player 1 distributes the contents of bowl 6 in anti-clockwise successive bowls, dropping one stone at a time. The dropped stones are depicted in white.

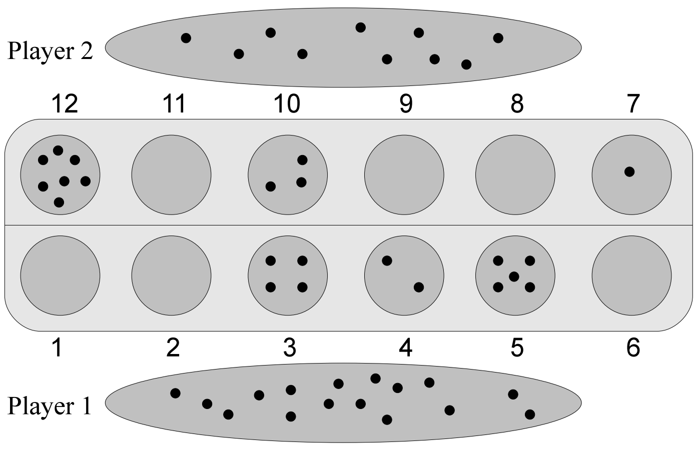

Figure 4: Player 1 captures the stones in all bowls that contained just one stone before the move, but which now contain multiple stones. Thus 8 stones are moved into player 1's capture pot.

- **Example:** Suppose the game has reached the configuration shown in Figure 2. It is player 1's turn to move and they have chosen to move the stones in bowl 6. They therefore pick up the seven stones in this bowl and place one stone in each of bowls 7, 8, 9, 10, 11, 12 and 1. The result of this is depicted in Figure 3, where the freshly dropped stones are shown in white. Observe that bowls 8, 9, 11 and 1 each contained one stone before the move, and now each contain two stones. Thus, player 1 captures the eight stones contained in these bowls, and moves them into their capture pot. The result of this is shown in Figure 4. This concludes player 1's move, and play passes to player 2.
- The game is over when one of the players has captured at least 24 stones. There are some rare situations where the game continues forever, but for the purposes of your program you can assume that all games will end with a clear winner.

What To Do

As this exercise is not assessed by your PPTs, and is to be written from scratch, there is no provided repository. A suggested design follows:

Class `Player`

The `Player` class represents a player in a game of Owari and thus primarily wraps a field that tracks the player's score. Furthermore, the following methods (here defined in terms of their postconditions) should be written to permit the score's update and retrieval:

Constructor `Player()`
Creates a new `Player` with an initial score of 0.

Method `void addToScore(int points)`
Adds `points` to the `Player`'s score.

Method `int getScore()`
Returns the `Player`'s current score.

Class `Bowl`

An instance of the `Bowl` class represents a single bowl in the game, which contains an integer number of stones. The methods of the `Bowl` class capture the appropriate subset of the game's rules:

- A bowl may either be emptied completely (`takeAllStones()`) or supplied with one additional stone at a time (`depositStone()`).
- Since a turn's score can only be calculated by knowing the initial number of stones in each bowl, each stone added by `depositStone()` should not update the contents of the bowl; this should only happen once the turn is logged as completed (`updateAndGetScore()`).

The class will therefore contain two fields: the first will store the number of stones in the bowl and the second will track the number of pending deposits. These will be manipulated by the public methods of the class, given below:

Constructor `Bowl(int stones)`
Creates a new `Bowl`, initially filled with the given number of stones.

Method `int getStones()`
Returns the number of stones in the `Bowl`.

Method `int takeAllStones()`
Empties the `Bowl` and returns the number of stones that were in it.

Method `void depositStone()`
Records the deposit of a new stone in the bowl.

Method `int updateAndGetScore()`
Signals the end of the bowl's involvement in a turn. Any pending deposits are used to update the number of stones in the bowl and return the acquired score (if appropriate).

Class `Game`

The `Game` class manages a game of Owari. It should contain fields for the players, the board and whose turn it is. Methods should be provided for checking and making moves, displaying the board and querying a game's state:

Constructor `Game()`
Creates a new `Game`.

Method `int getCurrentPlayer()`
Returns the identifier of the current player (e.g. `1` or `2`).

Method `void swapPlayers()`
Changes the current player.

Method `boolean isValidMove(int bowl)`
Returns `true` if and only if is legal for the current player to play the stones from the given bowl.

Method `boolean canCurrentPlayerMove()`
Returns `true` if and only if the current player has at least one legal move (verifiable with the `isValidMove()` method).

Method `void move(int bowl)`
: Updates the state of the game to reflect the current player moving with the stones in the given bowl. A necessary precondition is that the bowl is on the player's side of the board and is non-empty.

Method `int getLeadingPlayer()`
: Returns the identifier of the leading player (i.e. the player with the highest score).

Method `boolean isOver()`
: Returns `true` if and only if the game is over.

Method `void display()`
: Prints the game board to the console. You should include the scores of each player. For example, at the start of the game, your output may resemble that below:

```
Player 2: 0

  12    11    10    9     8     7
( 4 ) ( 4 ) ( 4 ) ( 4 ) ( 4 ) ( 4 )

( 4 ) ( 4 ) ( 4 ) ( 4 ) ( 4 ) ( 4 )
  1     2     3     4     5     6

Player 1: 0
```

Class `Owari`

The `Owari` class is the application's entry point. It therefore exports a `main` method similar to those that you have written in your previous lab exercises. Unlike your previous exercises, however, it will use instances of the classes you have implemented up to this point in order to manage an interactive game of Owari.

Method `static void main(String[] args)`
: Starts an interactive game of Owari. You should note that most of the required functionality is already present in the `Game` class – you need only handle input/output and the sequencing of the gameplay:

1. Display the board and prompt the current player for their move, assuming that they are able to make one.
2. If the move supplied is valid, enact it; if not, inform the user and request a new move.
3. If the game is over, print the winner and exit. If not, swap the players and repeat from step 1.

You should use the **IOUtil** class made available in previous exercises to read input from the user.

Testing

You should write suitable tests for each of the `Player`, `Bowl` and `Game` classes that test pieces of functionality in isolation – whether or not `Player.getScore` behaves as expected after an invocation of `Player.addToScore`, for example.

Submission & Assessment

This exercise will be unassessed – however it is *strongly* recommended that you attempt it. A Java lecture in week 10 will be spent working through the solution to this exercise as revision for your Week 11 Java Practice Lexis Test.

Suggested Extension

As an optional challenge, consider extending your program with an AI that you can play against. To do so, you'll need to modify your solution to take into account the following:

- In addition to managing a game and keeping the score, your program will need to generate moves for one of the players, with the object of winning the game.
- To allow a program to play Owari against either a human player, the program must be able to take the role of either player.
- A program which takes the role of player 1 should generate the first move. It will then wait for the user to type in the next move which will have been generated by the opposing player which has taken the role of player 2. The program will then alternately generate moves and get moves from the keyboard until the game is over.
- After a move is generated by the program, the game should proceed as if a human had entered it (i.e. the board should be displayed, the scores updated, etc.).
- You will need to devise and implement a strategy to generate a move given the state of the board at any time.
- To deal with the cases where it is impossible to capture all the stones you may wish to count the number of moves that have been made since the last capture and declare the game over after a fixed number of moves were made without a capture.