Programming IT, AY 2020-2021

AE2: Snakes and Ladders

Please check FAQ at end of this doc!

Introduction

Snakes and ladders is a popular board game (find details here: https://en.wikipedia.org/wiki/Snakes_and_Ladders or play online here: https://toytheater.com/snakes-and-ladders/). In the game, 2 or more players take turns to roll a die and then move their piece forward the number of places they roll (from 1 to 6). If their new destination is at the bottom of a ladder, they move up the ladder to its highest point. If their new destination is at the top of a snake, they slide down the snake to its lowest point. The player who gets to the final square first (or beyond it) is the winner.

In this AE you will create the classes required for a Snakes and Ladders game.

The AE is split into various stages. It is not expected that everyone will complete all stages, and a maximum grade available for reaching each stage is shown to help you manage your effort.

Marking scheme is shown below, in each question marks will be awarded for:

- Functional correctness
- Sound OO design
- Clean, tidy, commented code

Stage 1: The Player and Square classes

A board will consist of an array of `Square` objects. Consider this board:

```
 49(    )   48( -3)   47(    )   46(    )   45(    )
 40(    )   41(    )   42(    )   43(    )   44(    )
 39(    )   38(    )   37(    )   36(    ) S 35(    )
 30( -1)   31(    )   32( -4)   33(  2)   34( -1)
 29( -4) E 28(    )   27(    )   26(    )   25(    )
 20(    )   21(    )   22(    )   23(    )   24(    )
 19(    )   18(  4)   17(    )   16(    )   15( -1)
 10(    )   11(    )   12(    )   13(    )   14(    )
  9(    )    8(    )    7(    )    6(    )    5(    )
  0(    )    1(    )    2(    )    3(    )    4(    )
```

It has 10 rows and 5 columns. The starting square (0) is drawn in the bottom left. The winning position is 49 (top left). There are two players 'E' and 'S'. 'S' is at position 35 and 'E' at position 28.
Each `Square` object consists of:

- an integer position (0 to 49),
- references to any `Player` objects that are currently at that square and

- an integer `delta`.

The integer `delta` indicates how far up or down the player should move if they land here (a positive number indicates a ladder – see for example, position 18, and a negative number denotes a snake – see for example position 32). If there is no snake or ladder starting at a square, `delta = 0`. `Square` objects need a `setDelta` method.

Squares should have a `toString` method that returns a string representation of the square consisting of player information (from the `toString` method) for all players currently at the square, the square position, and the value of `delta` in brackets (if `delta` is not zero).

Players have an attribute of type `char` that allows them to be identified. In the example above, these are 'E' and 'S'. The `toString` method of the `Player` class just returns this as a string.

To make things easier later on, `Player` objects should hold a reference to the square in which they are currently located.

Task1: create `Player` and `Square` classes that admit the functionality described above. In your `Player` class, include a `main` method that demonstrates the creation of a player, the creation of a square, the assignment of the `Player` to the `Square`, and the `toString` methods of both the player and the square.

[*4 marks*]

Stage 2: The Board class

The `Board` constructor should take integers representing the number of rows and columns. Players always start at position 0. The board should always be drawn so that the row containing the final position (49 in our example) is the top and the starting row (containing 0) is at the bottom. The board should have an attribute that is an array of `Square` objects. A `Board` object should also store references to all players playing the game. When a player is added to the game they should be placed in the square at position 0.

The board should have a `toString` method that returns a string representation of the board (making use of the `toString` method of the `Square` class).

The `Board` class should have helper methods that return the row and column of a particular position, and the position corresponding to a particular row, column pair.

The `Board` class should have a method for adding players. When players are added, the board should store a reference to them for future use and place them on the starting square.

The `Board` class should have a method that returns a reference to the `Square` object at any integer position.

The rows of the board should **wrap** (see example above). I.e. when a player reaches the end of a row, their next square should be immediately above them.

Task 2: create a `Board` class that admits this functionality. In the `Board` class, include a `main` method that creates a board with 10 rows and 5 columns. The `main` method should also create two players, add them to the board, and print the board.

[*5 marks*]

Note: The wrapping part is worth 1 mark. All other marks are available if you do not get the rows to wrap.

Stage 3: Moving

Now we start to play! Players should be given a `move` method. It should return a binary value which will be `true` if the move they make results in winning, and `false` otherwise. The `move` method will roll a die (example below) and move the player forward the relevant number of places. If a player's move ends up on a square with a delta, they should move forward or backward by the relevant number of spaces.

Task 3: Add `move` methods to the players and a method to the board (`takeTurns`) that loops through the players and calls each of their `move` methods. If a `move` method returns `true`, that player has won and `takeTurns` should immediately terminate, returning `true`. Otherwise it should return `false`.

[*5 marks*]

Rolling a die:

```
Random r = new Random();
int count = r.nextInt(6) + 1; // the +1 because we want values
//between 1 and 6 and not 0 and 5
```

Note: if you implement move that doesn't do the deltas, you will still be eligible for 3 of the 5 marks.

Stage 4: Playing

Task 4: Create a new class (`Play.java`) that will contain a `main` method that will create a board, some players and start a game, playing it until completion. In this `main` method add deltas to some of the squares (at least two snakes, and two ladders). Ensure that no snake or ladder ends where another snake or ladder starts (this can result in an infinite loop).

[*4 marks*]

Stage 5: Different types of player

Task 5: Create a subclass of your `Player` class (call it `HumanPlayer`) that, when the `move` method is called, writes a message to the standard output requesting an input integer and then waits until the player enters an integer between 1 and 6. (you might find this useful for testing other tasks). Show this player being used in the `main` method of your `Play` class.

[*4 marks*]

Submission instructions

- Submission will be via **moodle.**
- Deadline (**provisional**): 4:30pm on Monday 1st March 2021.
- Submit a single .zip file (***not .rar, .7z, etc)*** that includes your .java files.
- Make sure that your student number is in a comment at the start of **every** .java file you submit.
- Name the .zip file: XXXXXXX_ProgAE2.zip where XXXXXXX is your student ID.

- If you have placed your .java files within a package, please remove the line: `package <name>;` from the top of each .java file.

We reserve the right to deduct marks for failure to comply with these instructions. If you're not sure, ask!

FAQs

1. *Is it important to make flashy graphics for my board?* No. Functionality is what we're after here.
2. *Should I comment my code?* Yes. Your code is the only thing you will submit. We reserve the right to penalise because we cannot understand, even if the code works. Provide clear comments.
3. *Should I provide unit tests?* You can get 100% of the marks without them. However, writing tests will help you to get things working quicker. If you do write them, feel free to submit them.
4. *If I can't get something to work, what should I do?* We can sometimes award marks for you telling us (in comments) how you might get something to work so if something isn't working, tell us why you think that is.
5. *How long should I spend on this assessed exercise?* Hard to say. However, we find every year that students spend **way** too long on AEs.
6. *I want to extend my code more as programming practice, any suggestions?* Great! It's a good way to improve your programming. How about abstracting the die into its own class and then subclassing for different types? E.g. maybe a die that only rolls 1s and 6s? How about placing snakes and ladders randomly, or making a Swing GUI for the game? You won't get any extra marks, but you'll become a better programmer! **Don't submit these extra classes – we need a clean version that we can mark precisely**
7. *Will you post an example solution?* Yes. Once we have all the submissions and have completed the marking. This might be a while after the deadline (people get extensions for being ill etc.) so please be patient! We will devote an examples class to going over our model solution.