Simple Snakes and Ladders Game: A Development Walkthrough

James Montgomery

Revised: 23 April 2017

Overview

This document is a kind of development walkthrough that takes you through developing a new class to represent a `Die` (which you've likely done previously) and then using this to complete a basic Snakes & Ladders game.

Download the sample code for this exercise. The code, or at least one *possible* solution, to parts 1, 2 and 4, is in a folder with the same name. If you get stuck, look at the example solution. The code for Part 3 represents the *starting point* for that section.

Part 1: Developing the Die class

This was covered in Lecture 7 and the notes in some detail. We want to model game dice with differing numbers of sides and which take responsibility for randomly selecting their next 'upward facing' side. In this part you'll write a `Die` class and a `DieTester` class (which will contain a `main()` method that instantiates one or more `Die` objects and calls their methods to check that they behave as expected).

The `Die` class should have the following characteristics:

Attributes:

- how many sides/faces it has
- current side facing 'up' (its current value)
- a `java.util.Random` object to obtain random numbers (which will need to be `import`ed)

Behaviours:

- report (get) how many sides it has
- report its current face value
- roll a new value; this can also return the new value

Constructors:

- one that sets the number of sides and initial face value to sensible defaults
- an alternative that accepts (as a parameter) the number of sides

When writing the alternative constructor that accepts the number of sides, make sure that it enforces a sensible lower limit on the number of sides. For instance, the call `new Die(2)` should probably create a new `Die` with four or more sides (you can choose which in the code you write).

When you're done check out the example code for Part 1. It's probably not the *only* way to solve the problem of modelling a `Die` but illustrates some elements of good coding style.

Part 2: An improved Die class

Depending on how you implemented the `Die` class earlier you may have noticed there are a lot of literal values (unnamed constants) spread through the code. Also, there's a lot of similarity between the two constructors: both set the initial face value to the same value, and both instantiate a `Random` object. The main difference is that one of them accepts the number of die sides as a parameter.

Replace all literal numbers with named constants

Anywhere you see a literal `int` value in your code (like 4 or 6), replace it with a named constant. This will make these values easier to change, if we ever need to, and make it less likely that the wrong value will be written somewhere. Above the instance data declarations (for `numFaces`, etc.) declare and initialise some new variables preceded by the keyword `final` for each constant value you need, as in:

```
final int DEFAULT_FACES = 6;
```

In fact, constants like this, which will be the same for *every* `Die` object, can be turned into class constants (rather than a separate copy for each instance), as in:

```
static final int DEFAULT_FACES = 6;
```

And, finally, if the constant could be useful information in other parts of the program then it can be marked as `public`, whereas if it represents information that should be known only to the `Die` class, then it can be marked `private`. This would give the (rather long) constant declaration of:

```
public static final int DEFAULT_FACES = 6;
```

which means we are declaring a publicly accessible, constant integer value called `DEFAULT_FACES`, with the value of 6, and which belongs to the whole class of `Die` objects.

Using `this` in a constructor

Now to reduce the amount of code duplication between the parameterless constructor and the constructor that accepts the number of faces as a parameter. The keyword `this` can be used inside the code for a class to refer to the current object (i.e., `this` object). It can also be used as the first line of one constructor to call another, which is useful if the two constructors differ only slightly. In the case of the `Die`, the difference is the number of die faces.

Refer to the sample code for Part 2 to see `this` being used.

Part 3: The top-level algorithm for Snakes and Ladders

The classes that make up this program (which are mostly already implemented) are:

- `Die`: represents a game die (as described above)
- `Board`: A snakes and ladders game board, which knows how many squares there are, where the snakes and ladders start and end and what messages appear at the beginning of each.
- `SALGame`: A partially interactive snakes and ladders game that (after you've finished it) knows where two players are located on the board and can run multiple games in sequence (until the user doesn't want to play any more).
- `SnakesAndLadders`: Contains a `main()` method whose sole responsibility it to instantiate a `SALGame` object (which will then run the game).

Before opening the .java files in the Part 3 folder, consider the following top level algorithm that will be implemented in `SALGame`:

- create one `Die` object and one `Board` object
- declare `int` variables for player positions and the current player
- see if user wants to play
- while user wants to play
    - play one game
        - set both positions to 0
        - set current player
        - while the game is not over
            - current player makes a move
            - if game is not over (now)
                - swap players
        - announce winner
    - see if user wants to play again

On paper, write down how you would break up that algorithm into different sub-tasks (each of which would eventually become a method). Draw a structure chart that shows how these sub-tasks would be related. Next you'll see how we decided to break up the game. Your approach may be different, it may even be *better*.

The approach we took

Open `SALGame.java` to see how our approach resulted in a partial implementation of the game.

Identify subtasks (methods)

- Constructor: creates `Die` and `Board` objects, then calls `startPlaying()`
- `startPlaying()`: The outer loop (does the user want to play); calls `playAGame()` to play a single game
- `playAGame()`: inner loop (while game is not over)
    - calls `makeAMove()`, which makes one move for the current player and reports if the game is over now
    - (if the game is not yet over) calls `swapPlayer()`, which swaps the current player
- For "housekeeping" purposes:
    - `trace()`, which displays a given debugging message on screen
    - `setTracing()`, which controls whether debugging messages are displayed or not

Then write "stub" versions of the methods. Looking at the code you can see that the algorithm for each method is described by calls to `trace()`, so if we run the `SnakesAndLadders` program we'll see a description of what *should* happen when it's completed.

Identify data (instance variables)

What data need to be shared *between* methods?

- reference variables for the `Die` object and `Board` object
- "counters"/"tokens": the position on the board of each player
- the current player
- the tracing switch (the boolean value that indicates if tracing messages will be printed or not)

Identify data (local variables)

What data is needed only *within* the methods?

- `startPlaying()`
  - needs a variable to hold the user's response (do they want to play again?)
- `playGame()`
  - needs a variable (`boolean`) to indicate whether or not the game is over
- `makeAMove()`, needs variables to hold
  - the value that `Die` rolled (so it can print it to the screen)
  - the value of the last square on the `Board` (so it can tell whether the current player has reached the last square and won the game)
  - the `boolean` value it will return to indicate if the game has been won

Identify data (Parameters and Results)

What values are needed and/or produced?

- `makeAMove()`
  - Needs to 'know' which player will move
    - Pass this as a parameter
    - This is a cleaner solution than writing two methods (one to move player 1 and a very similar one to move player 2)
  - Needs to 'tell' `playGame()` whether the latest move has reached the last square (which will mean the game is over).
    - Use a boolean return value to do this

Inspecting the partially completed game

If you haven't already, open all the .java files in the Part 3 folder in DrJava. You'll see that `Die` and `SnakesAndLadders` are not very interesting, the first because you've already seen it and the second because all it does is create an object of class `SALGame`.

The `Board` represents a snakes and ladders game board, which knows how many squares there are, where the snakes and ladders start and end and what messages appear at the beginning of each. It has these public methods:

- `int moveOn(int start, int dist)`: returns the new square after moving `dist` places from square `start`
- `int getLastSquare()`: returns the number of the last square on the board (with a few changes the game board could be made customisable to have a different size)
- `void setTracing(boolean on)` and `void checkBoards()` are development-time debugging methods; there's also a `private void trace(String message)` method that prints the given message if tracing (debugging) messages are enabled.

You do not need to understand all the code in `Board` yet.

Completing the game

Using the details above and the text in the tracing messages, implement the top-level algorithm. After each change, re-run the program (remember to select `SnakesAndLadders.java` in DrJava before clicking Run, since `SnakesAndLadders` contains the `main()` method).

If you get stuck at any point, take a quick look at the completed `SALGame.java` file in the Part 4 folder.

Part 4: Compare your completed game with ours

The Part 4 folder contains the completed game. It's not the most fun game to play, since a human user only has control over whether a game is played (to completion) or not. Perhaps you'd like to give the user more control, or simply slow down the play a bit. If you'd like to add a delay between each move, add the following method to `SALGame.java` and call it after each move has been completed, with the delay in milliseconds you want.

```
private void pauseFor(long ms) {
    try {
        Thread.sleep(ms);
    } catch (InterruptedException e) {
        System.err.println("Game interrupted while pausing");
    }
}
```

And if you have any questions about any part of the exercise then ask your tutor or the lecturer.
