Introduction to Computer Programming (CS 1323)
Project 5

Instructor: Andrew H. Fagg

October 21, 2014

Introduction

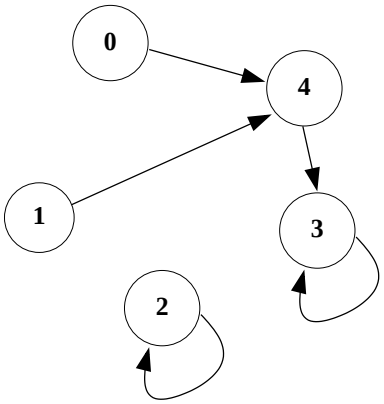
A *Finite State Machine* (FSM) is an abstract model of computation that allows us to formally describe (for limited cases) the idea that a program moves through a series of steps as it computes some output. FSMs are part of the theoretical study of *what is computable.* These models are also used in many different applications, from control of vending machines and traffic lights to managing the dialog that your computer has with a remote web server as it fetches a web page. Generalizations of FSMs are used to solve planning problems from a set of taxis trying to figure out whom to pick up next to robots planning how to cook dinner.

In this project, we will implement a simple form of a Finite State Machine that is defined by:

- A finite set of states, numbered 0, 1, ... $N-1$, and
- A *transition function* that defines for each state, what the next state will be.

At any one time, a Finite State Machine is in exactly one state (never zero and never more than one). At each step, the FSM transitions from its current state to the "next" state in the sequence. On the next step, the FSM transitions again (and so on).

The following figure illustrates a FSM with a total of $N = 5$ states. States are represented with labeled circles. Within the circle is the number that corresponds to that state. The arrows represent the transition function. If the FSM starts in state 0, it will transition to 4 on the first step, 3 on the next step, and then 3 again. Once this FSM reaches state 3, it never leaves. If the FSM starts in state 2, then its next state is 2, and it never leaves.



One way to represent the transition function encoded by the arrows in this figure is through a table. For this FSM, the transition function is:

State	**Next State**
0	4
1	4
2	2
3	3
4	3

There is one row in the table for each arrow in the figure. The order of the rows in this table is not important. The important property is (for our limited definition of a FSM) that each state appears exactly once in the **State** column. A state can appear any number of times in the **Next State** column. It is an error if there is a **Next State** that does not also appear in the **State** column.

In this project, we will implement this simple model of a FSM. The input to your program will be:

1. The numbers in the **Next State** column
2. The String "DONE"
3. A number that represents the starting state

The program will then respond with the sequence of states that the FSM visits. This sequence will end after a state is repeated. The program will then return to step 3 above and repeat.

Note: for full credit, you may assume FSMs that only have loops of length 1 (a state transitioning back to itself). We will not have FSMs that involve longer loops (e.g., 1 $\rightarrow$

$2 \rightarrow 3 \rightarrow 1$). However, extra credit will be awarded if you also address FSMs with longer loops (see the rubric below).

Objectives

By the end of this project, you will be able to:

1. create and manipulate **ArrayLists**, and
2. write a loop that traverses through an ArrayList in an unordered fashion.

Project Requirements

1. Write a **getNonNegativeInt()** method that takes as input a String prompt and a Scanner, and returns an integer. This method first prompts the user for the integer. If the entered value is a **non-negative** integer, then it is returned. If the entered value is a negative integer, then the user must be re-prompted. If the entered input is the string "done" (with any capitalization), then this method must return Integer.MIN_VALUE. Otherwise, the user must be re-prompted. Any re-prompting continues until a valid value is received.

2. Write a **traverseFSM()** method that takes as input an ArrayList of Integers that encodes the transition function and an integer start state. This method prints out the sequence of states that are visited by the FSM, starting with the start state and ending with the first repeated state.

   If the start state is invalid or if an intermediate state is invalid, then this method must print an error and return. *Invalid* here is defined as a state index that does not fall within the range of the table.

3. Write a **main()** method that prompts the user for the entries in the transition table, starting with state zero and ending when getNonNegativeInt() returns Integer.MIN_VALUE for the first time.

   This method then repeatedly prompts the user for a starting state number and prints out the sequence of states that result from this starting state. This process continues until the user enters "done".

Examples

Here are some example inputs and corresponding outputs from a properly executing program. You should carefully consider the cases with which you test your program. In particular, think about the key "boundary cases" (the cases that cause your code to do one thing or another, based on very small differences in input). User inputs are shown in bold.

Example 1

Please enter the transition for state 0: **4**
Please enter the transition for state 1: **4**
Please enter the transition for state 2: **2**
Please enter the transition for state 3: **3**
Please enter the transition for state 4: **3**
Please enter the transition for state 5: **DONE**
Please enter the starting state: **0**
State Sequence:
0
4
3
3
Please enter the starting state: **1**
State Sequence:
1
4
3
3
Please enter the starting state: **3**
State Sequence:
3
3
Please enter the starting state: **2**
State Sequence:
2
2
Please enter the starting state: **Done**

(program ends)

Example 2

Please enter the transition for state 0: **2**
Please enter the transition for state 1: **6**
Please enter the transition for state 2: **5**
Please enter the transition for state 3: **3**
Please enter the transition for state 4: **5**
Please enter the transition for state 5: **6**
Please enter the transition for state 6: **6**
Please enter the transition for state 7: **3**
Please enter the transition for state 8: **3**
Please enter the transition for state 9: **8**
Please enter the transition for state 10: **doNE**
Please enter the starting state: **0**
State Sequence:
0
2
5
6
6
Please enter the starting state: **1**
State Sequence:
1
6
6
Please enter the starting state: **9**
State Sequence:
9
8
3
3
Please enter the starting state: **10**
Initial state is not valid
Please enter the starting state: **foo**
Try again: **6**
State Sequence:
6
6

Please enter the starting state: **7**
State Sequence:
7
3
3
Please enter the starting state: **dOnE**

(program ends)

Example 3

Please enter the transition for state 0: **0**
Please enter the transition for state 1: **4**
Please enter the transition for state 2: **0**
Please enter the transition for state 3: **1**
Please enter the transition for state 4: **7**
Please enter the transition for state 5: **0**
Please enter the transition for state 6: **adfad**
Only non-negative values are allowed: **-5**
Only non-negative values are allowed: **2**
Please enter the transition for state 7: **afafa**
Only non-negative values are allowed: **DONe**
Please enter the starting state: **3**
State Sequence:
3
1
4
Reference to non-existent state 7. Terminating...
Please enter the starting state: **5**
State Sequence:
5
0
0
Please enter the starting state: **6**
State Sequence:
6
2
0

0
Please enter the starting state: **1**
State Sequence:
1
4
Reference to non-existent state 7. Terminating...
Please enter the starting state: **afdad**
Only non-negative values are allowed: **-10**
Only non-negative values are allowed: **7**
Initial state is not valid
Please enter the starting state: **100**
Initial state is not valid
Please enter the starting state: **3**
State Sequence:
3
1
4
Reference to non-existent state 7. Terminating...
Please enter the starting state: **2**
State Sequence:
2
0
0
Please enter the starting state: **DONE**

(program ends)

Project Documentation

Follow the same documentation procedures as we used in project 1. In particular, you must include:

- Java source file documentation (top of file)
- Method-level documentation
- Inline documentation

For full credit, you must execute Javadoc on your source file and include the results (in the *doc* directory) in your zip file.

Hints

Implement and test your program one method at a time (we call this *incremental development*). Convince yourself that each method is working properly before you move on to the next one.

The project requirements define specific method names, parameter types and return types. In order to receive full credit for the project, you must adhere to these requirements.

Project Submission

This project must be submitted no later than 2:00pm on Tuesday, October 28th to the project 5 D2L dropbox. The file must be called *Project5.zip* and be generated in the same way as for Projects 1..4.

Code Review

For the office hour sessions surrounding the project deadline, we will put together a "Doodle Poll" through which you can sign up for a 5-minute code review appointment. During this review, we will execute test examples and review the code with you. If you fail to show up for your reserved time, then you will forfeit your appointment and you must attend at a later time that is not already reserved by someone else.

If either the instructor or TA are free during office hours (or another mutually-agreed upon time), then we are happy to do code reviews, as well as provide general help.

We will only perform reviews on code that has already been submitted to the dropbox on D2L. Please prepare ahead of time for your code review by performing this submission step.

Code reviews must be completed no later than Tuesday, November 4th. If you fail to do a code review, then you will receive a score of zero for the project.

Anyone receiving a 95 or greater on Project 4 may opt to do an "offline" code review (which does not require your presence). Send the instructor email once you have submitted your project in order to schedule an offline review.

Rubric

The project will be graded out of 100 points. The distribution is as follows:

Implementation: 35 points

Program formatting: 15 points

(15) The program is properly formatted (including indentation, curly brace and semicolon locations).

(8) There is one problem with program formatting.

(0) The program is not properly formatted.

Data types and method calls: 10 points

(10) The program is using proper data types and method calls.

(5) There is one error in data type or method call selection.

(0) There are multiple errors in data type and method call selection.

Required Methods: 10 points

(10) All of the required methods are implemented correctly.

(0) The required methods are not implemented correctly.

Proper Execution: 30 points

Output: 15 points

(15) The program passes all tests.

(12) The program fails one test.

(9) The program fails two tests.

(6) The program fails three tests.

(3) The program fails four tests.

(0) The program fails five or more tests.

Execution: 15 points

(15) The program executes with no errors (thrown exceptions).

(8) The program executes, but there is one minor error.

(0) The program does not execute.

Documentation and Submission: 35 points

Project Documentation: 5 points

(5) The java file contains all of the required documentation elements at the top of the file.

(3) The java file is missing one of the required documentation elements.

(2) The java file is missing two of the required documentation elements.

(0) The java file is missing more than two of the required documentation elements.

Method-Level Documentation: 10 points

(10) Every method contains all of the required documentation elements ahead of the method prototype, and the Javadocs have been generated and included in the submission.

(7) The method documentation is missing one of the required documentation elements.

(3) The method documentation is missing two of the required documentation elements, or the Javadocs have not been submitted.

(0) The method documentation is missing more than two of the required documentation elements.

Inline: 10 points

(10) Every method contains appropriate inline documentation.

(7) There is one missing or incorrect line of inline documentation.

(3) There are two missing or incorrect lines of inline documentation.

(0) There are more than two missing or incorrect lines of inline documentation.

Submission: 10 points

(10) The correct zip file name is used.

(0) An incorrect zip file name is used.

Bonus: 6 points For each unique and meaningful error in this project description that is reported to the instructor, a student will receive an extra two points.

Extra Credit: 10 points If your program properly handles termination in the case of a loop that is longer than one state in length (e.g., $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$), then you may receive up to ten extra points.

References

- Finite State machines: http://en.wikipedia.org/wiki/Finite-state_machine