Assignment 3

CS 1027
Computer Science Fundamentals II

Due date: November 14 at 11:55 pm

Learning Outcomes

In this assignment, you will get practice with:

- Stacks
- Lists
- Interfaces
- Generics
- Advanced Algorithms
- Advanced Debugging

Introduction

A popular single-player card game is solitaire. In the game, there are many stacks of cards: the regular stacks (sometimes referred to as the tableau), and four initially empty foundation stacks that are later built up out of ascending consecutive cards of each suit. To win a game of solitaire, all cards must be moved to these foundation stacks. This assignment will revolve around a simplified solitaire that uses Domino tiles instead of cards.

A set of **Domino** tiles consists of 28 rectangular game tiles (see Fig 1.) Each tile has a combination of two numbers which is unique to that tile. In a normal set, the highest number is six and the lowest is zero or simply blank.

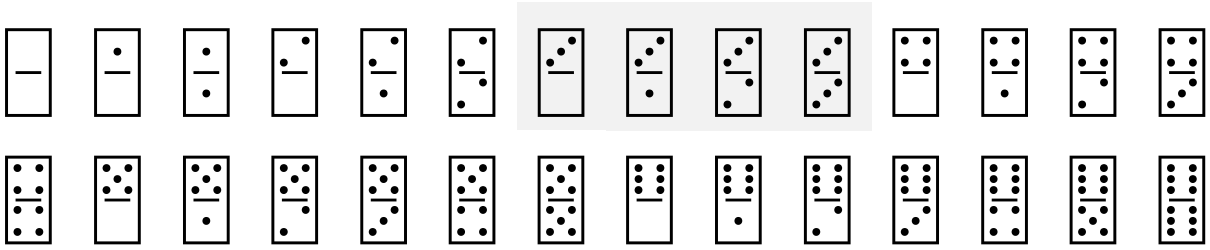


Fig 1. A complete set of Dominoes (with 6 as the largest number). The tiles containing a 3 as the highest number are highlighted—they will be considered an example of a set that starts with a double [3:3] and goes down through [3:2] and [3:1] to [3:].

In our simplified solitaire, called **DomSolitaire,** there will be an equal number of regular and foundation stacks with the exact number being determined by the largest number used. For instance, in Fig 2., there are four stacks of each type with the largest tile number used being 3. Each of the foundation stacks starts empty and will then get filled with tiles descending from doubles containing only tiles with that particular doubled number as its highest number.

Assignment 3

CS 1027
Computer Science Fundamentals II

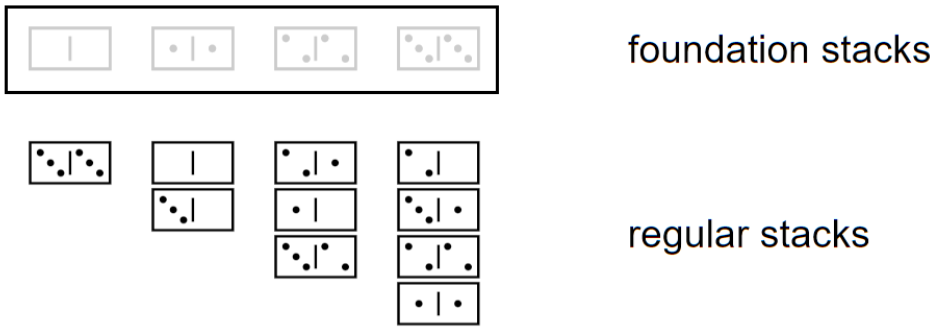
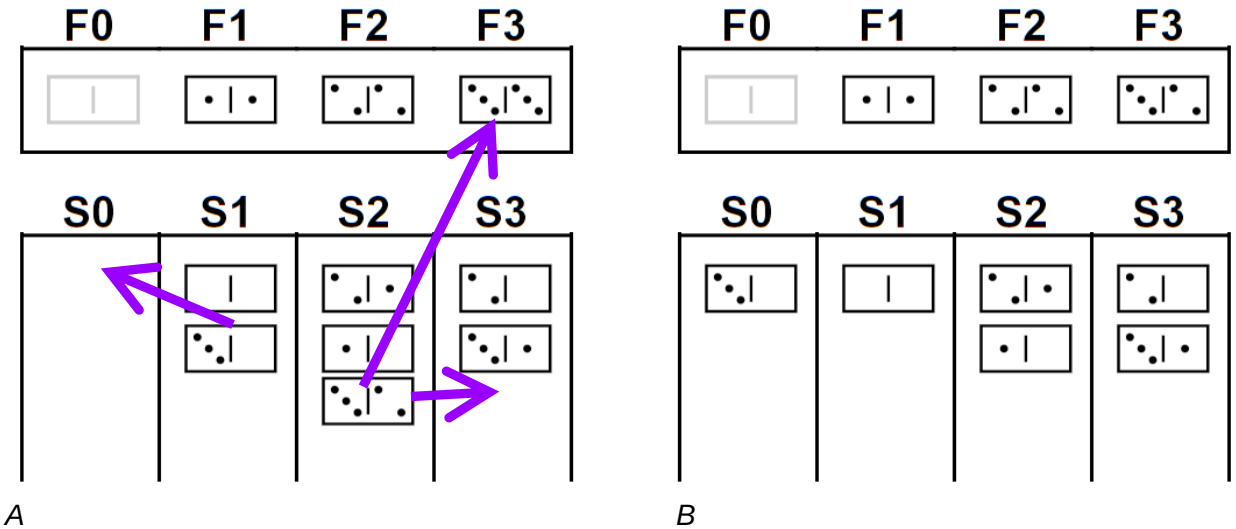


Fig 2. A DomSolitaire with 3 as it's biggest number. Notice that there are 4 empty foundations stacks and 4 regular stacks with sizes 1, 2, 3, and 4. If 4 was the biggest number used there would be 15 total tiles and 5 of each type of stack.

There are three types of moves: a move from a regular stack to a foundation stack, a move from a regular stack to a non-empty regular stack, and a move from a regular stack to an empty regular stack. Each of these moves is only permissible under certain conditions. A tile, T, on regular stack can move to a foundation stack either if T is a double tile which means that it can move to empty foundation or if T's low number (e.g. [3:1] 's low number is 1) is one number below the low number of the top of the foundation corresponding to T's high number (e.g [3:1] would have go to the foundation 3 and the [3:2] tile would have to be at the top of F3 for it to be able to move there). The next types of move only use the regular stack. A tile, T1, on top of a regular stack can move on to a non-empty stack with a tile, T2, on top if T1 and T2 share a number in common and T1's other number is one up from that T2's other number. The last type of move occurs if there is an empty regular stack. Any tile T, from that top of a regular stack with a blank can be moved to the empty regular stack. Fig. 3 shows some examples of permissible.



A *B*

Fig 3. (A)Three types of permissible moves: A tile from a regular stack to a foundation e.g. move S2 to F3 is permissible since the [3:2] tile is one down from the [3:3] tile (also [3:2]'s high number is a 3 it belongs in the F3 stack); S2 to S3 is permissible since [3:2] shares a 3 with the S3's [3:1] and it is one up; and S1 to S0 is permissible since S0 is empty and S1's top has a blank.(B) Shows the result of the first and third moves.

The purpose of these moves is to completely empty the regular stacks having moved all the tiles into the foundation stacks: see Fig 4.

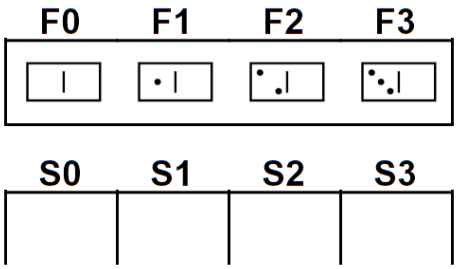


Fig 4. The final winning state of a game. F0 has 1 tile, F1 and 2 tiles ([1:1], [1:]). F2 has 3 tiles ([2:2], [2:1], [2:]), and F3 has 4 tiles ([3:3], [3:2], [3:1], [3:]) whereas all the regular stacks S1-S4 are empty.

To better illustrate the moves and the game, a complete game using is shown in Fig. 5. In this game, there are only 6 tiles comprising of tiles that have 2 as there largest number. The game take 9 moves: 6 moves from regular stacks to foundation (AEDFGHI), 2 moves from regular stacks to non-empty regular stacks(CE), and 1 move from a regular stack to an empty regular stack(B).

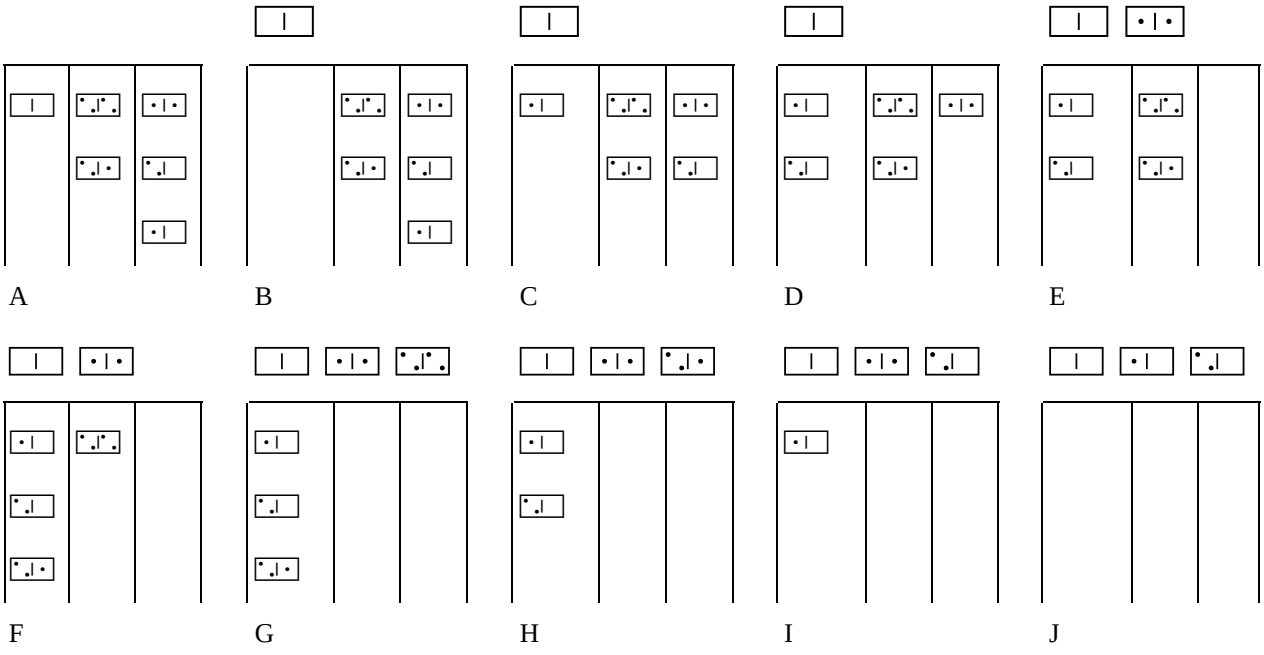


Fig. 5 A complete game with the Domino tiles with no number greater than 2

Assignment 3

CS 1027
Computer Science Fundamentals II

In this assignment, you are implementing a DomSolitaire game solution finder. Your code will attempt to find a sequence of moves that takes the initial setup with empty foundations and filled regular stacks to one with filled foundations and empty regular stack. There are two dimensions that determine the initial setup. One dimension is the number of Domino tiles used 3, 6, 10, 15, 21, and 28 which correspond to the highest number used on the tiles (1,2,3,4,5, and 6). For example, when 3 is the highest number there are 10 tiles as seen in Fig. 2. The more tiles tend to make the game more difficult. The particular set of tiles used in each game is determined at the start (see Domino.getSet(n)—for example getSet(3) was used for Fig. 2-4 and getSet(2) for Fig. 5) Another dimension is the initial ordering of the tiles in the regular stacks. The initial ordering is also determined at the start (See Domino.shuffle(seed) for details). Although many setups have solutions (are winnable), some setups will have no solutions.

Provided files

- ListADT.java
- UnorderedListADT.java
- UnorderedList.java
- StackADT.java
- EmptyCollecionException.java
- LinearNode.java
- StackArray.java
- Named.java ** interface for debugging
- StackLL.java ** note: implements Named
- **Domino.java** ** note: see above
- DomSolTests.java

The DomSolTests.java file will **help** to check if your java classes are implemented correctly. A *similar* file will be incorporated into Gradescope's auto-grader. **Passing all the tests within DomSolTests.java does not necessarily mean that your code is correct.**

Classes to Implement

For this assignment, you must implement two Java classes: **Move** and **DomSolitaire**. Follow the guidelines for each one below.

In of these classes, you may implement more private (helper) methods and it is encouraged for readability and reducing duplication of code. However, you may not implement more public methods **except** public static void main(String[] args) for testing purposes (this is allowed and encouraged). You may not add instance variables other than the ones specified in these instructions nor change the variable types or accessibility (i.e. making a variable public when it should be private). Penalties will be applied if you implement additional instance variables or change the variable types or modifiers from what is described here.

Assignment 3

CS 1027
Computer Science Fundamentals II

Move.java

Move is a simple class that holds information about a particular move in a sequence of moves. It will also hold a name to potentially be useful in debugging.

The class must have these private instance variables:

- private StackADT<Domino> from
- private StackADT<Domino> to
- private boolean completed
- private String name

The class must have the following public methods:

- public Move(StackADT<Domino> from, StackADT<Domino> to) [constructor]
  - calls the other constructor with name “m”
    - hint: use this(from, to, "m");
- public Move(StackADT<Domino> from, StackADT<Domino> to, String name) [constructor]
  - sets the all the corresponding instance variables and sets completed to false
- public void doIt()
  - pop an element in *from* and push it on *to*
  - sets *completed* to true
- public void undoIt()
  - pop an element in *to* and pushes it on *from*
  - sets *completed* to false
- public boolean isCompleted()
  - getter for *completed*
- public boolean equals(Object obj)
  - returns false if obj is not an instance of Move
  - returns true if this from and to are the same as obj’s from and to (in terms of memory location)
- public String toString()
  - constructs a string to represent this objects instance variable
    - {name}{from}->{to}{? or !}
      - {name} is the instance name
      - {from} either the *from’s* toString() unless from implements Named in which case {from} is from’s getName()
      - {to} either the *to’s* toString() unless from implements Named in which case {to} is *to*’s getName()
      - {? or !} is ? if *competed* is false or ! if *completed* is true
    - Examples:
      - `mStack |top>[3:1] <|->S3?`
        - “m” is the name of this move
        - “**Stack |top>[3:1]**” is the toString of from which must b a StackADT which is not a Named
        - “S3” is the name of to (which is a Named StackADT)
        - “?” means that this move is not completed
      - `testS0->F3!`
        - “test” is the name of this move

- "S0" is the name of from (which is a Named StackADT)
- "F3" is the name of to (which is a Named StackADT)
- "!" means that this move is completed

DomSolitaire.java

This class is used to represent game stack and find the solution sequences. Since, the nature of this class may cause problems while developing the code, there may be a need to follow the progression of how things are changing. To help see what is going on you can add lines that look like

```
if (debug) System.out.println("method 1: current state "+ this);
```

that can be activated by setting debug to true.

The class must have these private instance variables:

- private StackADT<Domino>[] foundation
- private StackADT<Domino>[] stack
- private String name
- private boolean debug

The class must have the following public methods:

- public DomSolitaire(int highestNum, int seed, String name) [constructor]
  - sets up arrays of StackADT for both stack and foundation and fills them with empty StackLL<Domino> with the names "S0" through "S{highestNum}" and names "F0" through "F{highestNum}"
  - creates a new Domino set of the correct size, shuffles them using the seed, and then adds them to the regular stacks so that
    - the first stack, stack[0] with the name "S0", gets the first Domino
    - the second stack, stack[1] with the name "S1", gets the next two Dominoes
    - etc.
- public String getName()
  - getter method for name
- public void setDebug(boolean debug)
  - setter for debug (debugging flag to print out intermediate results)
- public void reset(int seed)
  - empties the games stacks and creates a new Domino set of the correct size and shuffles them using the seed and adds them to the regular stacks so that
    - the first stack, stack[0] with the name "S0", gets the first Domino
    - the second stack, stack[1] with the name "S1", gets the next two Dominoes
    - etc.

- public boolean winner()
  - returns true if the current state of the stacks and foundations represent a win
    - false otherwise
  - For simplicity, we can assume only permissible moves were used to reach the current state. So, if the regular stacks are empty this DomSolitaire is a winner.
- public UnorderedListADT<Move> findSFMoves()
  - returns a List containing all the permissible moves from the current state that go from a regular stack to a foundation stack
- public UnorderedListADT<Move> findSESMoves()
  - returns a List containing all the permissible moves from the current state that go from a regular stack to an empty regular stack
- public UnorderedListADT<Move> findSSMoves()
  - returns a List containing all the permissible moves from the current state that go from a regular stack to a non-empty regular stack
- public void findMoves(StackADT<Move> st)
  - adds moves from findSESMoves(), findSSMoves(), and findSFMoves() to st in that order: so that the last moves added / pushed will be ones that go to a foundation (if any) and the first moves pushed will be the ones that go to empty regular stacks
- public Move createMove(String from, String to)
  - returns a Move created by from and to Strings and linking them to corresponding regular and foundation StackADTs
  - This method is used for initial tests in DomSolTests.java and is intended to be used for debugging.
  - Example createMove("S0", "F2") returns a new Move(stack[0], foundation[2])
    - Given the limited use, it is safe to assume that the first character will be either an "S" or an "F" (hint use String's startsWith("S"), charAt(0), or substring(0,1))
    - Similarly, to extract the number, there are a number of methods:
      - Rely on the fact that it's a single digit and use charAt(1)
      - Get the string that comes after the first character with substring(1) and then use Integer.parseInt("3")
- public String toString()
  - returns a String representing the current state of this DomSolitaire
    - The name of this DomSolitiaire followed by names and sizes of each foundation and followed by all the stack's toStrings (in the example below the name is "DomSolName")

```
DomSolNAME F0 0/1  F1 0/2  F2 2/3
S0 |top>[ : ] <|
S1 |top>[1:1] <|
S2 |top>[1: ] [2: ] <|
```

- public String showNamedContent()
  - returns a String representing the current state in a condensed format using Dominos names rather than the toStrings
    - just the foundation sizes followed by the stack's showNamedContent

```
000|A |C |B D |
```

- public StackADT<Move> findSolution(int maxSteps)
    - returns a Stack containing a sequence of Moves that would solve this game if a solution was found
    - returns an empty stack if there is no solution
    - returns null if the algorithm took more steps than maxSteps
    - This method uses the following exhaustive algorithm

The algorithm relies on a central stack of moves which is initialized by findMoves and will altering the stacks of domino, *stack* and *foundation* as indicated below. The algorithm starts processing the stack of moves provided:

- the moves stack is not empty
- the game is not in a winning state
- and the maxSteps has not been exceeded

Each processing step consists of examining the top element of the moves stack:

- If the move has not been completed
    - do the move
    - find more moves with findMoves and add them to the moves stack
- If the move has been completed
    - backtrack
    - remove the completed move from the moves stack

If the algorithm is no longer processing steps:

- If the maxSteps has been exceeded, return null
- otherwise
    - transfer all the completed moves into a new stack such that the moves appear in the reverse order from the order that they in the moves stack.
    - Return the new stack (if it is empty not solution was found)

Marking Notes

Functional Specifications

- Does the program behave according to specifications?
- Does it produce the correct output and pass all tests?
- Are the classes implemented properly?
- Does the code run properly on Gradescope (even if it runs on Eclipse, it is **up to you** to ensure it works on Gradescope to get the test marks)
- Does the program produce compilation or run-time errors on Gradescope?
- Does the program fail to follow the instructions (i.e. changing variable types, etc.)

Assignment 3

CS 1027
Computer Science Fundamentals II

Non-Functional Specifications

- Are there comments throughout the code (Javadocs or other comments)?
- Are the variables and methods given appropriate, meaningful names?
- Is the code clean and readable with proper indenting and white-space?
- Is the code consistent regarding formatting and naming conventions?
- Submission errors (i.e. missing files, too many files, etc.) will receive a penalty of 5%
- Including a "package" line at the top of a file will receive a penalty of 5%

Remember you must do all the work on your own. Do not copy or even look at the work of another student. All submitted code will be run through similarity-detection software.

Submission (due Monday, November 14 at 11:55 pm)

Assignments must be submitted to Gradescope, not on OWL. If you are new to this platform, see these instructions on submitting on Gradescope.

Rules

- Please only submit the files specified below.
- Do not attach other files even if they were part of the assignment.
- Do not upload the .class files! Penalties will be applied for this.
- Submit the assignment on time. Late submissions will receive a penalty of 10% per day.
- Forgetting to submit is not a valid excuse for submitting late.
- Submissions must be done through Gradescope. **If your code runs on Eclipse but not on Gradescope, you will NOT get the marks! Make sure it works on Gradescope to get these marks.**
- You are expected to perform additional testing (create your own test harness class to do this) to ensure that your code works. We are providing you with some tests, but we may use additional tests that you haven't seen before for marking.
- Assignment files are NOT to be emailed to the instructor(s) or TA(s). They will not be marked if sent by email.
- You may re-submit code if your previous submission was not complete or correct, however, re-submissions after the regular assignment deadline will receive a penalty.

Assignment 3

CS 1027
Computer Science Fundamentals II

Files to submit

- Move.java
- DomSolitaire.java

Grading Criteria

Total Marks: [20]

Functional Specifications:

[2] Move.java

[5] DomSolitaire.java

[10] Passing Tests (some additional, hidden tests will be run on Gradescope)

Non-Functional Specifications:

[1] Meaningful variable names, private instance variables

[1] Code readability and indentation

[1] Code comments