Metropolitan State University
ICS 240 - 50: Introduction to Data Structures
Spring 2022

Homework 04 – The `Stack` data structure
Total points: 40
Out: Tuesday, June 28, 2022
Due: 11:59 PM on Monday, July 12, 2022
Last day to submit with penalty: 11:59 PM on Thursday July 14, 2022

Objective

In this assignment, you are asked to implement a linked-list-based stack, called `ThingLinkedStack`, which is a Stack of Things. You will then implement static methods to use that stack.

IMPORTANT: You may NOT use the Stack from the `java.util.Stack` library. Instead, you must implement your own `Stack` as explained below.

Question 1: Stack

Part 1: Implement `ThingLinkedStack`

In this part, you will use/implement the following classes.

- `Thing`: use the same `Thing` that you created for the previous assignments and make sure to look for the requirements below and change your previous implementation as necessary.
- `ThingNode`: use the `ThingNode` class that you implemented in assignment 3.
- `ThingStackInterface`: this interface includes the same methods as IntStackInterface but the headers of the methods are changed to reflect that elements in the stack are of type `Thing` instead of int.
- `ThingLinkedStack`: in class, we implemented `IntLinkedStack` that stores integers. Start from the implementation of `IntLinkStack` and change as appropriate to implement the `ThingLinkedStack` which is a stack that can hold elements of type `Thing`.
- `ThingStackDriver`: includes a `main` method to test the functionality of your stack. This file also includes additional methods that use the stack as described below.

`class Thing`

You should use the same `Thing` class that you used in assignments 2 and 3 but make sure all the following requirements are met.

- Has exactly 2 instance variables such that:
  - one instance variable must be an integer (i.e, `int`)
  - one instance variable must be of type `String` and this variable will be used for comparing two Things. We will call this the search key.
- Getters and setters for all instance variables.
- An implementation of the `toString()` method where a `Thing` is represented in one line with the individual instance variables separated by tabs.
- Implement the `equals()` methods to test the equality of two things where two things are equal when they have the same values in **both** of the instance variables and String test should be case insensitive.
- Implement the `Comparable` interface and include the `compareTo()` method. Note that `Thing`s in your collection are to be **compared** based on the String search key first, then on the `int` instance variable and String comparison should be case insensitive.
- **Note 1:** Do not name your class `Thing.java`. Give your class a name that is indicative to the type of elements that you are maintaining. Also, the name of your class should be singular not plural (e.g, `Car` not `Car`**s**) because this class is used to represent only one element.

`class ThingNode`

This class represents a node in a linked list. Use the same `ThingNode` class that you implemented in Homework 3. The Node class should:

- hold objects of type `Thing`.
- include two private instance variables:
  - a `Thing` data element
  - a link of type `ThingNode`.
- include a constructor that takes two input parameters and uses them to initialize the two instance variables.
- Getters and setters for the instance variables
- The following static methods. You must change the methods as appropriate.
  - `display, listPosition, listLength,`and `listSearch.`

`class ThingStackInterface`

This class includes the same methods as the `IntStackInterface` but the headers of methods are changed to reflect that the data stored in the stack are of type `Thing` instead of `int`.

`class ThingLinkedStack`

- `ThingLinkedStack` is a linked-list-based stack that can hold elements of type `Thing`. Start from the `IntLinkedStack` implementation and change as appropriate.
- The class has exactly 2 private instance variables such that:
    - one instance variable must be `ThingNode` reference variable and this variable will be used to keep track of the top of the stack.
    - one instance variable must be a counter that keeps track of the number of items on the stack.
- The class has the following methods:
    - a constructor that sets the top of stack to null and initializes the number of items on the stack to 0.
    - `isEmpty()` returns true if there are no elements on the stack and returns false if the stack has one or more data elements.
    - `peek()` returns a copy of the element stored at the top of the stack. `peek()` does not change the stack in any way . `peek()` throws an `EmptyStackException` if the stack is empty.
- `pop()` removes and returns the element stored at the top of the stack. `pop()` throws an `EmptyStackException` if the stack is empty.
- `push()` is a void method with one parameter which is an element of type `Thing` to be added to the top of the stack.
- `toString()` returns a String which lists the elements in the stack from top to bottom. `toString()` does not change the stack.
- `size()` returns the number of items on the stack.

Part 2: Implement `ThingStackDriver`

First, add the `main` method and write statements to test all the methods that you implemented in `ThingLinkedStack`. Then, implement the methods described below as **static** methods in `ThingStackDriver` and write code, in the `main` method, to call and test the methods. Note the following:

- You must use the `ThingLinkedStack` class that you implemented in part 1. **Do NOT use the Java-provided Stack.**
- All the following methods are **static** and are implemented in the `ThingStackDriver` class.
- Include appropriate comments for each method to briefly explain your algorithm.

Method	Description	Suggested Test Cases
`stackToInt()`	A static method that takes a stack of `Thing`s as a parameter. Assume that the input stack contains 7 or less things and all of them has an integer attribute in the range 0 to 9. The method returns an integer number representation of the values stored on the stack where the most significant digit is at the top of the stack. For example, if the input stack is as shown,	*4 elements in the stack. *empty stack *1 element *1 element with integer attribute 0.

	the output integer is 5478. When the method has finished, the input stack will be empty. Top of Stack: Red,5 Blue,4 Green,7 Red, 8	
`popSome()`	A static method that takes two input parameters, a `ThingLinkedStack` and an integer value `count`. The method then pops `count` elements from the stack. The method returns the sum of the integer attributes of the popped elements. If the stack has less than count values, the method returns -1. The input stack is changed because some elements are popped.	*Stack has 3 things and `count` = 3. *Stack has more than 3 things and `count` = 3. *Stack has less than 3 things and `count` = 3. *Stack has no thing and `count` = 3. *Stack has several things and `count` = 0. *Stack has several things and `count` is negative.
`extractFromStack()`	A static void method that takes two input parameters, a `ThingLinkedStack` and a `Thing` target element. The method removes all occurrences of the input target. All other elements in the stack must remain unchanged and in the same order. You must use the `Thing` class' `equals()` method.	*Stack has multiple things, one of which matches the input target. *Stack has multiple things, several of which match the input target. *Stack has multiple things, none of which match the input target. *Stack is empty.
`equalStacks()`	A static method that takes two `ThingLinkedStack`s as input parameters, and returns `true` or `false` based on whether the two stacks are equal or not. Assume that two stacks are equal if they have the same `Thing`s in the same order. <u>The two input stacks must remain unchanged after the call to this method</u>. The method must use the `Thing` class `equals()` method.	*Stacks are not empty and are equal. *Stacks are not empty and are not equal. *One stack is empty (returns false). *Both stacks are empty (returns true). *Test for case insensitivity.

Other Requirements

- All instance variables are private. Use getters and setters as appropriate.
- Use the exact same name (including upper case and lower case letter) for all methods as specified in the above description.
- Follow the method requirements in terms of the number, data type, and order of input parameters and the output data type.
- The program is robust with no runtime errors or problems.
- Test your code thoroughly. Suggested test cases are provided above for the Driver class. However, you should test *all* the classes thoroughly.
- Use good coding style throughout including
  - correct private/public access modifiers
  - consistent indenting
  - meaningful variable names
  - normal capitalization conventions
  - other aspects of easy-to-read code
- Document the source of any borrowed code
- Provide a UML structure diagram for this program.

Important Tips

- Start early and plan plenty of time.
- Solve and test one piece at a time.
- Solve the problem in stages. Save working versions periodically so you can return to a version that works as needed.
- Use the exact same name for methods (case-sensitive) as explained above.
- It is always better to receive partial credit for a program which runs but lacks one or more features than to receive no credit for a program that does not compile nor run.
- If you submit a partial implementation, your driver must include test cases for what you have completed or you will not receive credit.

Submission - Important

For this homework, you need to upload a **zip** file that includes the following five files:

- `Thing.java`
- `ThingNode.java`
- `ThingStackInterface.java`
- `ThingLinkedStack.java`
- `ThingStackDriver.java`

Upload **only one** **.zip** file to the D2L folder called Homework 4. **It is important that you upload your code in only one zip file. Your assignment will not be graded if you upload individual files to the drop box.**