CMSC 420: Spring 2023

Programming Assignment 0: Adjustable Stack

Handed out: Tue, Jan 31. Due: **Wed, Feb 8 (11:59pm)**.

Overview: This is a start-up project designed to acquaint you with the programming/testing environment and submission process we will be using this semester. This will involve only a small bit of data structure design and implementation.

The Adjustable Stack: In this assignment we will implement a generalization of the simple stack data structure, but along the way we will introduce a useful concept, called a *locator*. Like the standard Java stack data structure, our data structure is generic and stores objects of a given type, called `Element`.

```
public class AdjustableStack<Element> { ... }
```

It implements the standard stack operations of `push`, `pop`, and `peek` (which returns the top entry without removing it). In addition, we allow entries within the stack to be adjusted by either *promotion* or *demotion*. Promoting moves an entry one position closer to the top by swapping it with the entry immediately above it. Demoting moves it down one position by swapping it with the entry just below it. We will also support a *depth* operation, which returns how far an entry is from the top.

In order to implement these last three operations, we need an efficient mechanism for referring to specific entries of the stack. (Java provides a search operation, but this is very slow, since it involves searching the entire stack contents.) We do this using an object called a *locator*. A locator identifies an specific entry in the stack. This is implemented as a public class, called `Locator`, that is internal to the `AdjustableStack` class. Whenever an object is pushed into the stack, the `push` operation returns a locator referring to this entry. The promotion, demotion, and depth operations are each given a locator referencing the object on which to apply the operation.

How do we do this? Let's assume we implement the stack in the standard manner as an array of entries, along with an index `top`, which indexes the top entry of the stack. When an item is pushed on the stack, we create a new locator object, which stores the index of the entry in the stack array. Then, when we wish to promote or demote an entry, we provide the locator in order to identify this entry. An skeletal example is shown below.

```
public class AdjustableStack<Element> {
    public class Locator {
        private int index;
    }
    public Locator push(Element element) { ... }
    public void promote(Locator loc) { ... }
}
```

Here is an example of how this might be used. We create a new stack of strings, push three entries, and then we promote the middle entry.

```
AdjustableStack<String> stack = new AdjustableStack<String>();
AdjustableStack<String>.Locator loc1 = stack.push("cat"); // stack: cat
AdjustableStack<String>.Locator loc2 = stack.push("dog"); // stack: dog cat
AdjustableStack<String>.Locator loc3 = stack.push("pig"); // stack: pig dog cat
stack.promote(loc2); // promote dog. New stack: dog pig cat
```

The contents of the stack after the three pushes is shown in Fig. 1(a). The result after promoting `"dog"` is shown in Fig. 1(b). Observe that when `"dog"` and `"pig"` exchange places their associated locators are updated accordingly.

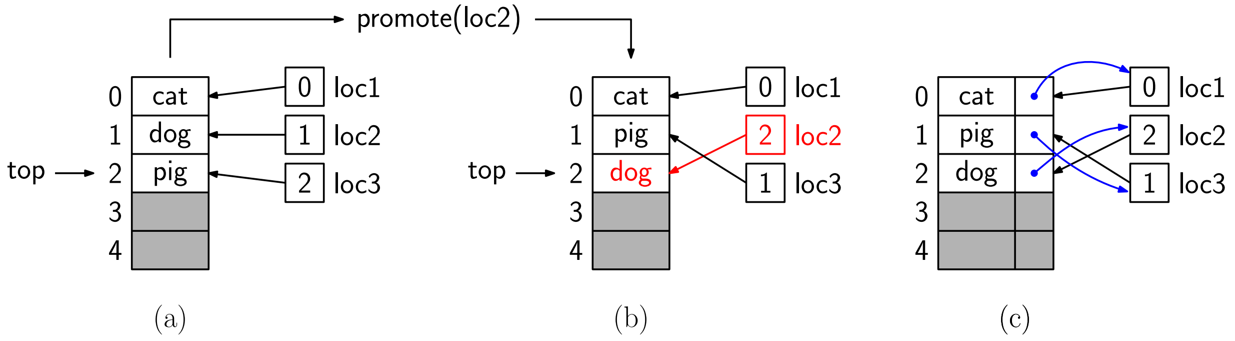


Figure 1: Locators and promotion.

You might wonder, why do we go through the extra effort of creating this special `Locator` class. Couldn't the `push` operator simply return the integer index of the newly inserting entry? The problem is that as other entries are promoted and demoted, and given items position in the stack will change dynamically. (As shown in Fig. 1(b), the locator for `"pig"` had to be changed, even though it was not named in the promotion opertion.) The purpose of the locator is to keep track of where the item is at all times.

But, you may see an obvious problem. If an entry's position is changed because one of its neighbors is promoted, how to we find its associated locator? To make this work efficiently, in addition to the locator referencing an entry in the stack, we need each stack entry to reference the associated locator. This way, when the entry moves, we can update the locator. This is illustrated in Fig. 1(c). You could do this in one of two ways. You could have two parallel arrays, one an array of type `Element`, containing the stack contents, and the other of type `Locator`, containing references to the locators. Alternatively, you could create a new (private) inner class within `AdjustableStack`, which contains two members, the element and the locator. Either approach is fine.

Operations: Here is a list of all the public methods your program is to implement.

`AdjustableStack()`: This creates a new empty stack. (`Hint:` While it is tempting to use Java's built-in `stack` object, this is not a good idea because you will need to implement your own `push` operation. It is inconvenient to use a fixed-size array, since you cannot predict in advance how many entries we will push, which necessitates expanding the array. We would recommend that you store your stack in an expandable array, such as Java's `ArrayList`. This allows you to efficiently access individual elements, and you can add as many entries as needed.)

`Locator push(Element element)`: This pushes `element` onto your stack. It also creates a new `Locator` object that references this element and returns this `Locator`.

`Element pop()`: This removes the element from the stack and returns its value. If the stack is empty, this throws an `Exception` with the message `"Pop of empty stack"`.

`Element peek()`: This returns a reference to the top element of the stack, without altering its contents. If the stack is empty, this throws an `Exception` with the message `"Peek of empty stack"`.

`int size()`: Returns the number of elements currently in the stack.

`void promote(Locator loc)`: Promotes the stack entry referenced by `loc`. If `loc` references the top of the stack, then this does nothing. Otherwise, it swaps this entry with the entry that is one position closer to the top of the stack. You may assume that that given locator is valid, and it refers to an element that is currently in the stack.

`void demote(Locator loc)`: Demotes the stack entry referenced by `loc`. If `loc` references the bottom entry of the stack, then this does nothing. Otherwise, it swaps this entry with the entry that is one position farther from the top of the stack. You may assume that that given locator is valid, and it refers to an element that is currently in the stack.

`int getDepth(Locator loc)`: Returns the depth of the stack entry referenced by `loc`. This is defined to be the number of positions from the top of the stack (so, the depth of the top itself is defined to be zero). You may assume that that given locator is valid, and it refers to an element that is currently in the stack.

`ArrayList<String> list()`: This returns a Java `ArrayList` whose members are the elements of the stack, listed from the top of the stack down to the bottom. For example, if you started with an empty stack and performed `push("cat"); push("dog"); push("pig")`, this returns an `ArrayList` containing ⟨`"pig", "dog", "cat"`⟩.

What you need to do: We will provide you with two programs that take care of the input and output (`Part0Tester.java` and `Part0CommandHandler.java`). All you need to do is to implement the above functions. In fact, we will give you a skeleton program, `AdjustableStack.java`, with all the function prototypes, and you just need to fill them in.

```
package cmsc420_s23; // Do not alter this line
import java.util.ArrayList;

public class AdjustableStack<Element> {

    public class Locator { /* ... */ }

    public AdjustableStack() { /* ... */ }
    public Locator push(Element element) { /* ... */ return null; }
    public Element pop() throws Exception { /* ... */ return null; }
    public Element peek() throws Exception { /* ... */ return null; }
    public int size() { /* ... */ return 0; }
    public ArrayList<Element> list() { /* ... */ return null; }
    public void promote(Locator loc) { /* ... */ }
    public void demote(Locator loc) { /* ... */ }
    public int getDepth(Locator loc) { /* ... */ return 0; }
}
```

Sample input/output: Here is an example of what the input and output might look like.

Input:	Output:
`push:cat`	`push(cat): successful`
`push:dog`	`push(dog): successful`
`push:pig`	`push(pig): successful`
`list`	`list: pig dog cat`
`size`	`size: 3`
`promote:dog`	`promote(dog): successful`
`list`	`list: dog pig cat`
`depth:pig`	`depth(pig): 1`
`pop`	`pop: dog`

What we give you: We will provide you with skeleton code to get you started on the class Projects page (`Part0-Skeleton.zip`). This code will handle the input and output and provide you with the Java template for `AdjustableStack`. All you need to do is fill in the contents of this class. Note that directory structure has been set up carefully. You should not alter it unless you know what you are doing.

Files: Our skeleton code provides the following files, which can be found in the folder "`cmsc420_s23`". Note that all must begin with the statement "`package cmsc420_s23`".

`Part0Tester.java`**:** This contains the main Java program. It reads input commands from a file (by default `tests/test01-input.txt`) and it writes the output to a file (by default `tests/test01-output.txt`). You can alter the name of the input and output files.

▷ *You should not modify this except possibly to change the input and/or output file names. The output is sent to a file in the* `tests` *directory, not to the Java console. Also note that if you use* `Eclipse`*, the contents of the* `File Explorer` *window are not automatically updated. You will need to refresh its contents to see the new output file.*

We will provide you with a few sample test input files along with the "expected" output results (e.g., `tests/test01-expected.txt`). Of course, you should do your own testing. To check your results, use a difference-checking program like "diff".

Note that the tester program does not generate output to the console (unless there are errors). The output is stored in the output file in the `tests` directory.

`Part0CommandHandler.java`**:** This provides the interface between our `Part0Tester.java` and your `AdjustableStack.java`. It invokes the functions in your `AdjustableStack` class and outputs the results. It also catches and processes any exceptions.

▷ *You should not modify this file.*

Submission Instructions: Submissions will be made through Gradescope. There is no limit to the number of submissions you can make, and only the last submission will be graded. Here is what to do:

- Log into the CMSC420 page on Gradescope, select this assignment, and select "`Submit`". A window will pop up (see Fig. 2). Drag your file `AdjustableStack.java` into the window. If you generated other files, zip them up and submit them all. Select "`Upload`".

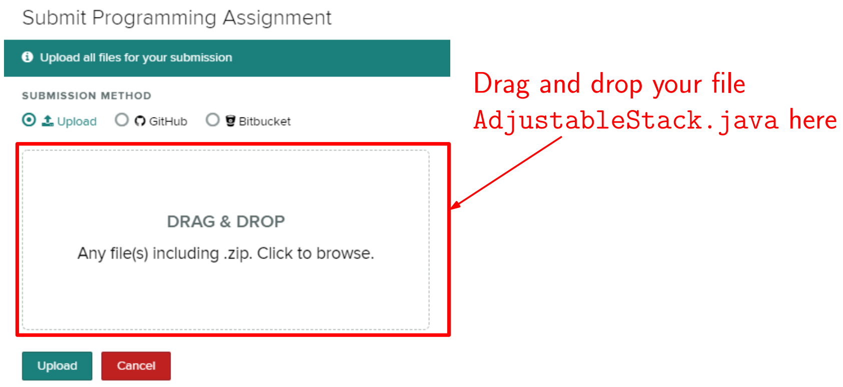
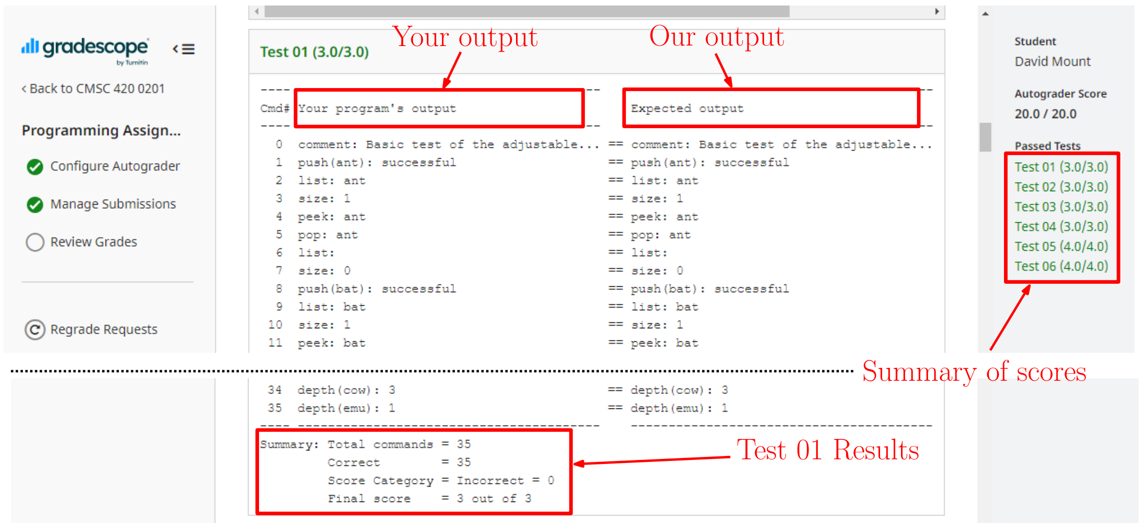

Submit Programming Assignment
Upload all files for your submission
SUBMISSION METHOD
Upload
GitHub
Bitbucket
DRAG & DROP
Any file(s) including .zip. Click to browse.
Upload
Cancel
Drag and drop your file AdjustableStack.java here

Figure 2: Gradescope submission. Drag your file `AdjustableStack.java` into the box.

- After a few minutes, Gradescope will display the results (see Fig. 3). Normally, a portion of your grade will depend on good style and efficiency, but for this initial program, only the autograder score will be used.



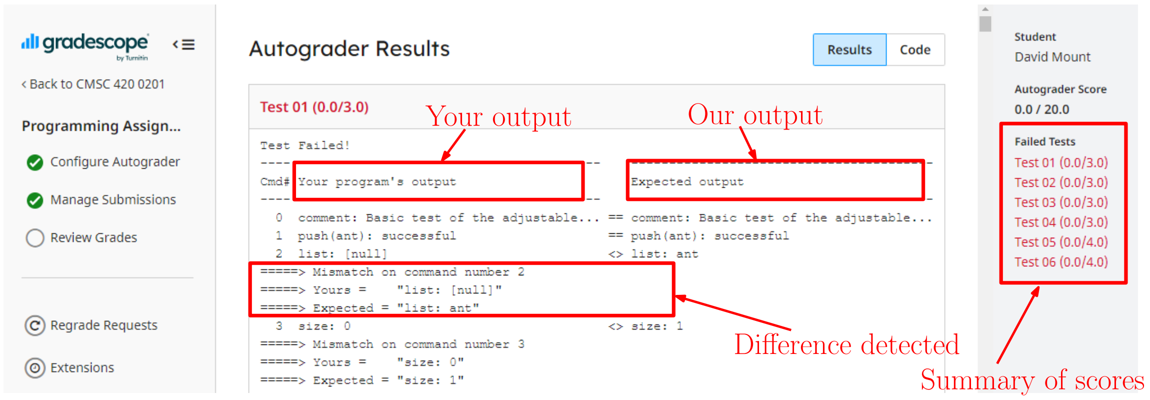
Test 01 (3.0/3.0)

```
---- ---------------------------------------   ---------------------------------------
Cmd# Your program's output                     Expected output
---- ---------------------------------------   ---------------------------------------
  0  comment: Basic test of the adjustable... == comment: Basic test of the adjustable...
  1  push(ant): successful                    == push(ant): successful
  2  list: ant                                == list: ant
  3  size: 1                                  == size: 1
  4  peek: ant                                == peek: ant
  5  pop: ant                                 == pop: ant
  6  list:                                    == list:
  7  size: 0                                  == size: 0
  8  push(bat): successful                    == push(bat): successful
  9  list: bat                                == list: bat
 10  size: 1                                  == size: 1
 11  peek: bat                                == peek: bat

 34  depth(cow): 3                            == depth(cow): 3
 35  depth(emu): 1                            == depth(emu): 1
---- ---------------------------------------   ---------------------------------------
Summary: Total commands = 35
         Correct        = 35
         Score Category = Incorrect = 0
         Final score    = 3 out of 3
```

Figure 3: Gradescope autograder results (correct).

- On the top-right of the page, it shows a summary of the scores of the individual tests as generated by the autograder. (If there are compilation errors, these will be displayed on this page.) The center of the window shows a line-by-line summary, with the output generated by your program on the left and the expected output on the right. If there are mismatches, these will be highlighted (see Fig. 4).
- The final score is based on the number of commands for which your program's output



Test 01 (0.0/3.0)

```
Test Failed!
-------------------------------------------  -------------------------------------------
Cmd# Your program's output                    Expected output
-------------------------------------------  -------------------------------------------
   0  comment: Basic test of the adjustable... == comment: Basic test of the adjustable...
   1  push(ant): successful                    == push(ant): successful
   2  list: [null]                             <> list: ant
=====> Mismatch on command number 2
=====> Yours =    "list: [null]"
=====> Expected = "list: ant"
   3  size: 0                                  <> size: 1
=====> Mismatch on command number 3
=====> Yours =    "size: 0"
=====> Expected = "size: 1"
```

Student
David Mount

Autograder Score
0.0 / 20.0

Failed Tests
Test 01 (0.0/3.0)
Test 02 (0.0/3.0)
Test 03 (0.0/3.0)
Test 04 (0.0/3.0)
Test 05 (0.0/4.0)
Test 06 (0.0/4.0)

Figure 4: Gradescope autograder results (incorrect).

differs from ours. Note that the comparison program is very primitive. It compares line by line (without considering the possibility of inserted or deleted lines) and is sensitive to changes in case and the addition of white-space.

Requirements: Since this is the first assignment, there are no requirements regarding efficiency or good coding style. The grade is based entirely on the Gradescope autograder.

Partial Credit: If you don't have time to implement the full version, you can get 50% partial credit by simply implementing the operations that do not involve locators, namely the constructor and the operations, `push`, `pop`, `peek`, and `size`.