# Course Description

**Weekly Overview**

This week is exam week.

# Institutional Learning Outcomes

**Main Objectives**

* Take the semester exam.

# Understand recursion, both mathematic

# Discipline Specific Outcomes

# Student Readings

None

**Daily Outline**

Day 1: Semester Exam

**Included Resources**

Teacher Exam Notes

Semester Exam: The Stack

Stack.java (student copy)

Stack.java (solution)

StackTest.java (JUnit Test)

**Semester Exam: Teacher Notes**

The semester exam will be to implement a Stack using an array. Students should have been given the assignment ahead of time, so they are free to think about it and work on it. They are not free to bring in any notes for the exam.

In order to implement the Stack properly, students will need to (1) keep track of the “top” of the stack using an index, and (2) resize the array once it is full, which involves setting up a secondary array and copying over the data.

We are going to give details on the initial construction, with an array of size 100, as well as instruction on resizing (adding an additional 100 slots). The reason for this is that we want students to *have* to resize their array. In other words, we want to avoid the very inefficient solution (in terms of memory) of starting with an absurdly larger (and empty) array. This is important to note because the JUnit test provided will not check this aspect of the code. This is because the data is to be private. Thus, in order to fully score the exam, you will need to exam student code.

At the start of the exam period, spend 15 minutes fielding any questions that they have about the nature of a Stack. Remember, the goal is to test student’s ability to write a class, not to understand a Stack. So feel free to explain this data structure so that everyone understands what is being asked. Field any questions about code expectations as well.

Explain to students that the code they will download has a JUnit test file. It is not important that they fully understand how this file works. They only need to know that they can compile it and press the “Run Tests” button in BlueJ. A window will pop up and explain any errors that might be in their class. If you have already discussed JUnit, this should not be a big deal. But even if you have not, it is a very simple thing to use. You may have to wal around during the exam period to make sure students know how to use it. The unit tests go through every scenario possible, including peeking at and popping an empty stack.

The students have the entire exam period to download the code, fill in the missing methods, and submit their code through GitHub. Assure them that even if it is not complete at the end, the code they have written will be scored. There can be some anxiety about “not finishing” or the code “not working.” Encourage them to make sure they attempt all the methods so that they do not get bogged down in getting one method “to work properly” while failing to provide any code for other methods. The best use of time is to first write the entire class. Then work on compiling, testing, and adjusting.

**Semester Exam: The Stack**

You semester exam will be to write a class for an object called a *stack*. The definition of a stack is:

1. A collection of data (yours will be Strings)
2. The data has a “top” entry
3. You can look at the top entry only (we call this “peeking” at the top of the stack)
4. You can remove the top entry, but only the top entry (we call this “popping” the top of the stack)
5. You can add to the top of the stack (we call this “pushing” an entry on to the top of the stack)
6. You can check the size of the stack

For example, consider the following code:

Stack s = new Stack ();

s.push(“Jake”); //”Jake” is now on the top of the stack

s.push(“Christina”); //”Christina” is now on the top of the stack

s.push(“Tatiana”); //”Tatiana” is now on the top of the stack

int j = s.size(); //j should be 3 because there are 3 things on the stack

String one = s.pop(); //one should be “Tatiana”, ”Christina” is now on top

String two = s.pop(); //two should be “Christina”, ”Jake” is now on top

s.push(“Kateri”); //”Kateri” is now on the top of the stack

s.push(“Jude”); //”Jude” is now on the top of the stack

s.push(“Benedict”); //”Benedict” is now on the top of the stack

j = s.size(); //j should be 4 (it started as 3, two thing removed, 3 added)

String three = s.pop(); //what is three?

s.push(“Sebastian”); //”Sebastian” is now on the top of the stack

s.push(“Thomas”); //”Thomas” is now on the top of the stack

s.push(“Edmund”); //”Edmund” is now on the top of the stack

s.push(“Marie”); //”Marie” is now on the top of the stack

s.pop(); //”Marie” is removed (not stored), //”Edmund” is now on top

s.pop(); //” Edmund” is removed (not stored), //”Thomas” is now on top

j = s.size(); //j should be 5 (count the pushes and pops)

String four = s.peek(); three is “Thomas” because “Thomas” is on the top

j = s.size(); //j should still be 5, and “Thomas” is still on top

In order to implement this, you will use a private array. However, a stack has a dynamic size that has not limit. You will have to think about how you can accomplish this, and you may need more private instance variables.

In addition to “getting this to work”, there are two operational requirements. These are non-negotiable:

1. You must initialize your array to 100 upon construction.
2. When the array become “full” and you need to resize, you should add 100, e.g. the first resize will make the array of length 200, the second resize will make the array of length 300, etc.

The reason for this is to be memory efficient. We *could* initialize the array to an absurdly large length to avoid having to resize, but these spots will remain “empty.”

You may think about this ahead of time and even start to code. However, you will be given a shell from which you must work, and you may not bring in any code or notes to the exam.

On exam day, you will also be given a file that you can use to test your stack as you go.

Good luck.

//Stack.java - Student Version

/\*\*

\* Stack with an array implementation

\* A Stack is a collect of data that has a "top" element.

\* Only the top element can be accessed.

\* When an element is added to the stack ("pushed"), the

\* new element becomes the top element.

\*

\* This implmementaiton uses an array to store the data.

\* A stack has no limit to the number of elements that can

\* be added, but an array does, so you will have to think

\* about how to deal with the structure when the underlying

\* array is full, but a push() is called.

\*

\* @author (Jake Tawney)

\* @version (January 7, 2019)

\*/

public class Stack{

private String[] data;

//you might need some extra private instance variables

public Stack(){

//You must initialize the array to length 100

}

/\*\*

\* This method pushes the String x onto the stack

\* After the method call, the "top" element is

\* the String x, and the size of the stack is one

\* more than the size before the push method was called.

\*/

public void push(String x){

//When resizing, you must do so by adding 100 to the length of the array

}

/\*\*

\* This method removes the top element from the stack and

\* returns it. After the method call, the size of the stack

\* is one less than it was prior to the method call.

\*

\* In the even that the stack is empty when pop() is called,

\* System.out.prinln("Error - Empty Stack"), and return

\* the null String.

\*/

public String pop(){

}

/\*\*

\* This method return the top element from the stack,

\* but does not remove it.

\* After the method call, the size of the stack

\* is the same as it was prior to the method call.

\*

\* In the even that the stack is empty when peek() is called,

\* System.out.prinln("Error - Empty Stack"), and return

\* the null String.

\*/

public String peek(){

}

/\*\*

\* This method returns the number of elements in the stack.

\* Note that this is different than the current capacity of

\* the underyling array.

\*/

public int size(){

}

}

//Stack.java

//Teacher Version (solution)

/\*\*

\* Stack with an array implementation

\* A Stack is a collect of data that has a "top" element.

\* Only the top element can be accessed.

\* When an element is added to the stack ("pushed"), the

\* new element becomes the top element.

\*

\* This implmementaiton uses an array to store the data.

\* A stack has no limit to the number of elements that can

\* be added, but an array does, so you will have to think

\* about how to deal with the structure when the underlying

\* array is full, but a push() is called.

\*

\* @author (Jake Tawney)

\* @version (January 7, 2019)

\*/

public class Stack{

private String[] data;

private int top;

public Stack(){

data = new String[100];

top = -1;

}

/\*\*

\* This method pushes the String x onto the stack

\* After the method call, the "top" element is

\* the String x, and the size of the stack is one

\* more than the size before the push method was called.

\*/

public void push(String x){

if(top + 1 >= data.length){

String[] newData = new String[data.length + 100];

for(int i=0; i<data.length; i++){

newData[i] = data[i];

}

data = newData;

}

data[top + 1] = x;

top++;

}

/\*\*

\* This method removes the top element from the stack and

\* returns it. After the method call, the size of the stack

\* is one less than it was prior to the method call.

\*

\* In the even that the stack is empty when pop() is called,

\* System.out.prinln("Error - Empty Stack"), and return

\* the null String.

\*/

public String pop(){

if(top == -1){

System.out.println("Error - Empty Stack");

return null;

}

String retString = data[top];

top--;

return retString;

}

/\*\*

\* This method return the top element from the stack,

\* but does not remove it.

\* After the method call, the size of the stack

\* is the same as it was prior to the method call.

\*

\* In the even that the stack is empty when peek() is called,

\* System.out.prinln("Error - Empty Stack"), and return

\* the null String.

\*/

public String peek(){

if(top == -1){

System.out.println("Error - Empty Stack");

return null;

}

return data[top];

}

/\*\*

\* This method returns the number of elements in the stack.

\* Note that this is different than the current capacity of

\* the underyling array.

\*/

public int size(){

return top + 1;

}

}

//StackTest.java

import static org.junit.Assert.\*;

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

import java.io.\*;

/\*\*

\* The test class StackTest.

\*

\* @author (Jake Tawney)

\* @version (January 16, 2019)

\*/

public class StackTest

{

private PrintStream sysOut;

private final ByteArrayOutputStream outContent = new ByteArrayOutputStream();

@Before

public void setUpStreams() {

sysOut = System.out;

System.setOut(new PrintStream(outContent));

}

@After

public void revertStreams() {

System.setOut(sysOut);

}

@Test

public void testConstructor()

{

// Setup

final Stack myStack = new Stack();

// Method under test

assertEquals(myStack.size(), 0);

assertEquals(outContent.toString(), "");

}

@Test

public void testPushAndPeek()

{

// Setup

final Stack myStack = new Stack();

myStack.push("Hello");

// Method under test

assertEquals(myStack.size(), 1);

assertEquals(myStack.peek(), "Hello");

assertEquals(outContent.toString(), "");

}

@Test

public void testPop()

{

final Stack myStack = new Stack();

myStack.push("Hello");

String y = myStack.pop();

// Method under test

assertEquals(myStack.size(), 0);

assertEquals(y, "Hello");

assertEquals(outContent.toString(), "");

}

@Test

public void testSize()

{

// Setup

final Stack myStack = new Stack();

int sizeTest = 50;

for(int i=0; i<sizeTest; i++)

myStack.push("Hello " + i);

// Method under test

assertEquals(myStack.size(), sizeTest);

assertEquals(outContent.toString(), "");

}

@Test

public void testPeekEmpty()

{

// Setup

final Stack myStack = new Stack();

// Method under test

assertNull(myStack.peek());

assertEquals(myStack.size(), 0);

assertEquals(outContent.toString(), "Error - Empty Stack\n");

}

@Test

public void testPopEmpty()

{

// Setup

final Stack myStack = new Stack();

// Method under test

assertNull(myStack.pop());

assertEquals(myStack.size(), 0);

assertEquals(outContent.toString(), "Error - Empty Stack\n");

}

@Test

public void testNumerousPushes()

{

// Setup

final Stack mySmallStack = new Stack();

int smallSize = 100;

for(int i=0; i<smallSize; i++)

mySmallStack.push("Hello " + i);

final Stack myMediumStack = new Stack();

int mediumSize = 150;

for(int i=0; i<mediumSize; i++)

myMediumStack.push("Hello " + i);

final Stack myLargeStack = new Stack();

int largeSize = 1050;

for(int i=0; i<largeSize; i++)

myLargeStack.push("Hello " + i);

// Method under test

assertEquals(mySmallStack.size(), smallSize);

assertEquals(myMediumStack.size(), mediumSize);

assertEquals(myLargeStack.size(), largeSize);

assertEquals(outContent.toString(), "");

}

@Test

public void testPushAndPop()

{

// Setup

final Stack mySmallStack = new Stack();

int smallSize = 50;

for(int i=0; i<smallSize; i++)

mySmallStack.push("Hello " + i);

String test1 = mySmallStack.pop(); //should be "Hello smallSize – 1"

String test2 = mySmallStack.pop(); //should be "Hello smallSize – 2"

String test3 = mySmallStack.pop(); //should be "Hello smallSize – 3"

mySmallStack.push("Hello A");

mySmallStack.push("Hello B");

String test4 = mySmallStack.pop(); //should be "Hello B"

String test5 = mySmallStack.pop(); //should be "Hello A"

String test6 = mySmallStack.pop(); //should be "Hello smallSize - 4"

String test7 = mySmallStack.pop(); //should be "Hello smallSize - 5"

mySmallStack.push("Hello C");

String test8 = mySmallStack.pop(); //should be "Hello C"

final Stack myLargeStack = new Stack();

int largeSize = 203;

for(int i=0; i<largeSize; i++)

myLargeStack.push("Goodbye " + i);

for(int j=0; j<10; j++)

myLargeStack.pop();

String test9 = myLargeStack.pop(); //should be "Goodbye largeSize – 11"

for(int k=0; k<20; k++)

myLargeStack.push("Goodbye Again " + k);

String test10 = myLargeStack.pop(); //should be "Goodbye Again 19"

for(int a = 0; a<20; a++)

myLargeStack.pop();

String test11 = myLargeStack.pop(); //should be "Goodbye largeSize - 13"

// Method under test

assertEquals(test1, "Hello " + (smallSize - 1));

assertEquals(test2, "Hello " + (smallSize - 2));

assertEquals(test3, "Hello " + (smallSize - 3));

assertEquals(test4, "Hello B");

assertEquals(test5, "Hello A");

assertEquals(test6, "Hello " + (smallSize - 4));

assertEquals(test7, "Hello " + (smallSize - 5));

assertEquals(test8, "Hello C");

assertEquals(test9, "Goodbye " + (largeSize - 11));

assertEquals(test10, "Goodbye Again 19");

assertEquals(test11, "Goodbye " + (largeSize - 13));

assertEquals(mySmallStack.size(), smallSize - 5);

assertEquals(myLargeStack.size(), largeSize - 13);

assertEquals(outContent.toString(), "");

}

@Test

public void fillAndEmpty()

{

// Setup

final Stack myStack = new Stack();

int mySize = 1025;

for(int i = 0; i<mySize; i++)

myStack.push("Hello " + i);

for(int j = 0; j<mySize; j++)

myStack.pop();

// Method under test

assertNull(myStack.peek());

assertEquals(myStack.size(), 0);

assertEquals(outContent.toString(), "Error - Empty Stack\n");

}

}