7.1 Stack abstract data type (ADT)

Stack abstract data type

A ***stack*** is an ADT in which items are only inserted on or removed from the top of a stack. The stack ***push*** operation inserts an item on the top of the stack. The stack ***pop*** operation removes and returns the item at the top of the stack. Ex: After the operations "Push 7", "Push 14", "Push 9", and "Push 5", "Pop" returns 5. A second "Pop" returns 9. A stack is referred to as a ***last-in first-out*** ADT. A stack can be implemented using a linked list, an array, or a vector.

PARTICIPATION ACTIVITY 7.1.1: Stack ADT.

Animation captions:

1. A new stack named "route" is created. Items can be pushed on the top of the stack.
2. Popping an item removes and returns the item from the top of the stack.

PARTICIPATION ACTIVITY 7.1.2: Stack ADT: Push and pop operations.

1) Given numStack: 7, 5 (top is 7).
Type the stack after the following push operation. Type the stack as: 1, 2, 3

Push(numStack, 8)

Check **Show answer**

2) Given numStack: 34, 20 (top is 34)
Type the stack after the following two push operations.
Type the stack as: 1, 2, 3

Push(numStack, 11)
Push(numStack, 4)

Check Show answer

3) Given numStack: 5, 9, 1 (top is 5)
What is returned by the following pop operation?

Pop(numStack)

Check Show answer

4) Given numStack: 5, 9, 1 (top is 5)
What is the stack after the following pop operation? Type the stack as: 1, 2, 3

Pop(numStack)

Check Show answer

5) Given numStack: 2, 9, 5, 8, 1, 3 (top is 2).
What is returned by the second pop operation?

Pop(numStack)
Pop(numStack)

Check Show answer

6) Given numStack: 41, 8 (top is 41)
What is the stack after the following operations? Type the stack as: 1, 2, 3

Pop(numStack)
Push(numStack, 2)
Push(numStack, 15)
Pop(numStack)

Check Show answer

Common stack ADT operations

Table 7.1.1: Common stack ADT operations.

Operation	Description	Example starting with stack: 99, 77 (top is 99).
Push(stack, x)	Inserts x on top of stack	Push(stack, 44). Stack: 44, 99, 77
Pop(stack)	Returns and removes item at top of stack	Pop(stack) returns: 99. Stack: 77
Peek(stack)	Returns but does not remove item at top of stack	Peek(stack) returns 99. Stack still: 99, 77
IsEmpty(stack)	Returns true if stack has no items	IsEmpty(stack) returns false.
GetLength(stack)	Returns the number of items in the stack	GetLength(stack) returns 2.

Note: Pop and Peek operations should not be applied to an empty stack; the resulting behavior may be undefined.

PARTICIPATION ACTIVITY 7.1.3: Common stack ADT operations.

1) Given inventoryStack: 70, 888, -3, 2
What does GetLength(inventoryStack) return?

- ○ 4
- ○ 70

2) Given callStack: 2, 9, 4
What are the contents of the stack after Peek(callStack)?

- ○ 2, 9, 4
- ○ 9, 4

3) Given callStack: 2, 9, 4
What are the contents of the stack after Pop(callStack)?

- ○ 2, 9, 4
- ○ 9, 4

4) Which operation determines if the stack contains no items?

- ○ Peek
- ○ IsEmpty

5) Which operation should usually be preceded by a check that the stack is not empty?

- ○ Pop
- ○ Push

CHALLENGE ACTIVITY | 7.1.1: Stack ADT.

422352.2723990.qx3zqy7

Start

Given numStack: 30, 80 (top is 30)
What is the stack after the following operations?

```
Pop(numStack)
Push(numStack, 12)
Pop(numStack)
Push(numStack, 84)
```

Ex: 1, 2, 3

1 2 3

Check Next

7.2 Stacks using linked lists

A stack is often implemented using a linked list, with the list's head node being the stack's top. A push is performed by creating a new list node, assigning the node's data with the item, and prepending the node to the list. A pop is performed by assigning a local variable with the head node's data, removing the head node from the list, and then returning the local variable.

PARTICIPATION ACTIVITY 7.2.1: Stack implementation using a linked list.

Animation content:

undefined

Animation captions:

1. Pushing 45 onto the stack allocates a new node and prepends the node to the list.
2. Each push prepends a new node to the list.
3. A pop assigns a local variable with the list's head node's data, removes the head node, and returns the local variable.

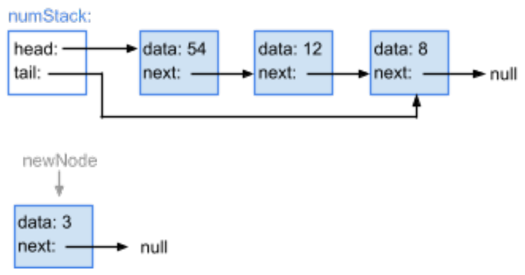
PARTICIPATION ACTIVITY 7.2.2: Stack push and pop operations with a linked list.

Assume the stack is implemented using a linked list.

1) An empty stack is indicated by a list head pointer value of _____.

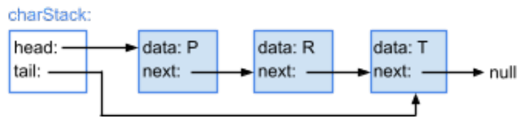
- ◯ newNode
- ◯ null
- ◯ Unknown

2) For StackPush(numStack, item 3), newNode's next pointer is pointed to _____.



- ◯ Node 54
- ◯ Node 12
- ◯ null

3) The operation StackPop(charStack) will remove which node?



- ◯ Node P
- ◯ Node R
- ◯ Node T

4) StackPop returns list's head node.

- ◯ True
- ◯ False

CHALLENGE ACTIVITY 7.2.1: Stacks using linked lists.

422352.2723990.qx3zqy7

Start

Given an empty stack numStack, what does the list head pointer point to? If the pointer is null, enter null.

1 2 3

Check Next

7.3 Lab 13 - Stack of Strings

Lab 13 - Stack of Strings

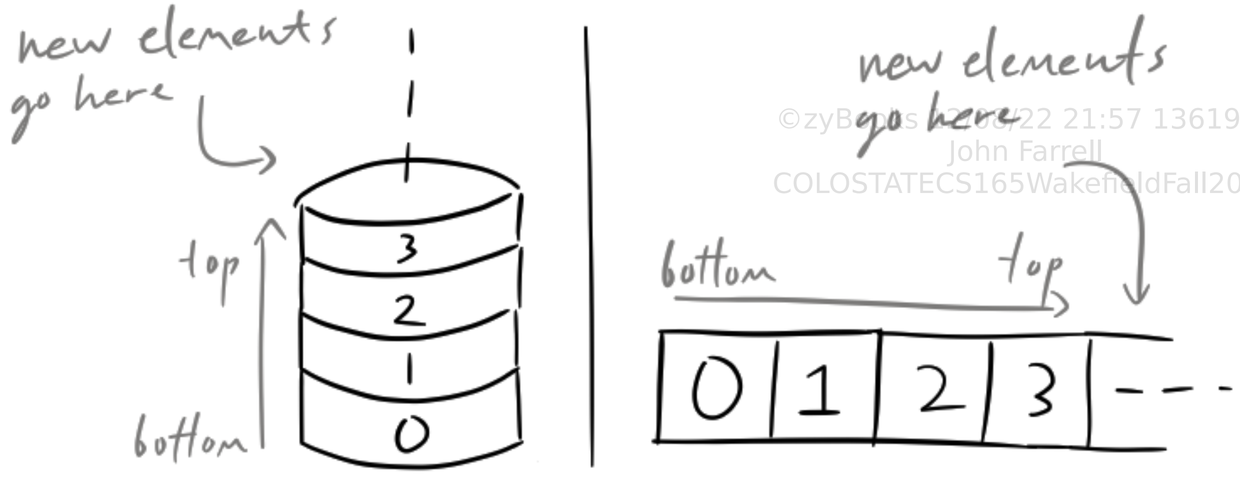
After getting through the Recursion module, you were probably all too familiar with the StackOverflowException. The name comes from how Java handles function calls. Each function that is invoked by your program is placed on a *stack*, so they can be kept track of in chronological order. If you call too many functions in a row without letting other functions resolve, too many things are put on the stack and it overflows, not unlike an overzealous waiter pouring too much water into a glass.

In this lab, we're going to create our very own stack, so we can experience for ourselves the thrill of putting too many things into it and watching it break! Just like with the ArrayList lab, we will start with a stack that only works with Strings, and move on to making a generic stack in the next lab.

Implementing a Stack

Implementing a stack turns out to be very similar to implementing an ArrayList. In fact, the way that Java's Stack class is implemented uses exactly the same technique as the ArrayList - it stores the data in an array that resizes when it gets too full. We're not going to have you implement the same resizing technique, however, as you've already done enough of that in the aforementioned ArrayList lab - the only important part is that **your stack should use an internal array**.

How exactly should you use the array? Stacks are quite simple data structures, so the intuitive way is probably the right way. Imagine your array as essentially being a stack laying on its side - the "top" of the stack is the right side, and the bottom is the left side.



In the image above, the left might be your intuitive mental image of the stack - a literal stack of elements - and the right is what it looks like as an array.

The comments in the source file will guide you through exactly what each function should do. **REMEMBER: capacity is the amount of elements a stack (or other object) can hold, and the size is the number of elements it contains.**

Exceptions

Your stack may run into errors, such as trying to remove an element from an empty stack or add to a full one. We ask you to use exceptions to deal with such errors in this lab. Recall that exceptions are a way to signal errors, and you can use a `throw` statement to tell Java that something wrong has happened:

```
throw new NoSuchElementException();
```

Remember that you don't have to *catch* these exceptions. It's the job of whoever is using the stack to account for that. Our job is only to complain when someone isn't using the stack properly.

When you're done writing each of the stack's methods according to the comments above them, try running the main method to test your code. If anything goes wrong, it will let you know through an error message. Be sure to read these in full, as they will tell you which methods you should look at to help debug your stack.

That's all! Get stacking, and make us something truly LEGO-worthy!

Testing

Make sure your code is functioning properly by running StringStack.java. You can download the

.java file and run it in your own environment or code right here in zyBooks. You will know your code is working properly if you get the following message:

```
All tests passed! Now go play some Jenga because you're the stacking
master.
```

Submission

In Submit mode, select "Submit for grading" when you are ready to turn in your assignment. In this lab, we are testing to see if your methods are functioning correctly.

422352.2723990.qx3zqy7

LAB ACTIVITY 7.3.1: Lab 13 - Stack of Strings 7 / 7

Downloadable files

StringStack.java **Download**

StringStack.java Load default template...

```
 import java.util.*;

/* YOUR CODE HERE
 * This entire class is just a skeleton for your code, plus a main method
 * for testing.
 */
public class StringStack {

    String[] nodeData;
    int size;

    /* You will need some data fields here - at the very least, some kind of
     * String array. You may also want to keep track of the size of the stack, i.e. the nu
     *REMEMBER: Capacity is how much the stack can hold, and size is the number of element
     */

    /* Puts the stack into a valid state, ready for us to call methods on.
```

Develop mode **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program Input (from above) → StringStack.java (Your program) →

Program output displayed here

Coding trail of your work What is this?

```
10/11 T-- W-0,7 min:15
```

7.4 Lab 12 - Generic Stack

Module 6: Lab 12 - Generic Stack

Strings Are So Last Year

This lab includes the following files:

```
L12
└── GenericStack.java
└── Calculator.java*
└── numbers.txt
```

* (This is the **main** file in zyLabs and is read-only because you do not need to modify anything.)

Here is the jar if you want to code in a different environment: L12.jar. Note: you likely will have to update the path to numbers.txt.

In the last lab, we made a stack that only works with Strings. Unfortunately, plain old Strings are quite boring, and we'd like our stack to work with even more general data types. This lab will have you not only make your stack so it works with any kind of data, but also fix some existing generic code to actually work with our new stack.

As with the last lab, **make sure you remember that size is not capacity. Size is the number of elements the stack contains, and capacity is how many elements it can hold.**

Make It Generic!

The first part of this lab is **making your stack generic**. Feel free to take the code from your working StringStack class - everything except the main method - and paste it into the GenericStack class. Change the name of the constructors to match the new name of the class, then modify the whole class so it uses generics, and can store any type that a programmer asks for.

Until you successfully complete this, the main method will give you nasty compiler errors. Once they stop, you should be good to move on!

Numbers and Readers

To take advantage of polymorphism, Java has a lot of inheritance hierarchies built-in. You may not have known that simple numeric classes, like **Integer** and **Float**, are actually part of one! Java contains a generic number class creatively called **Number**, and pretty much any class in Java whose purpose is to store numbers inherits from this class. (Note that this does not include the primitive `int`, `float`, and so on, since there are not classes)

Deep within Java's I/O packages, there's another interesting hierarchy of classes; the **Reader** and its many children. There are many types of Readers that can extract characters from different things, such as Strings, Files, and arrays.

Scrolls of Numbers

A few years back, I bought a box of random numbers from an intrepid salesman who insisted they were special. I'd like to find out if that's really the the case. Unfortunately, I've stored the numbers in a bunch of different places - some of them are in Strings, others are in files, it's a bit of a mess.

So I created a generic calculator that, using Readers, can read in a list of numbers from any source, put them into a stack, and then add them all up. Since I'm a good programmer, I put each of these steps in their own functions. Take a look at each of the methods and make sure you understand how they function.

`makeStack` is supposed to take in a Reader, of any kind, and return a stack of Numbers from the data in that Reader.
`evaluate` is designed to take a stack of Numbers and add all of the numbers together, returning the result as a double.
A note for this function: the Number class can't be directly added to a primitive (like a `double`, `int`, etc.) However, a Number object can be converted into a primitive with the use of the `doubleValue()` function.
I use the `parse` function to help with `makeStack`, and you can see how the return type matches the function and how it is used in `makeStack`.

Wrapping up

You'll know when you've successfully completed this lab once you can compile and run the code without errors, and you get output that looks like this:

```
76.4
76.39999961853027
4584425.0
15.324
```

Submission

In Submit mode, select "Submit for grading" when you are ready to turn in your assignment.

You are now armed with the knowledge necessary to tackle this lab. Go forth!

422352.2723990.qx3zqy7

LAB ACTIVITY | 7.4.1: Lab 12 - Generic Stack | 7 / 7

Downloadable files

`GenericStack.java` , `Calculator.java` , and `numbers.txt` **Download**

Current file: GenericStack.java Load default template...

```java
import java.util.*;

/* YOUR CODE HERE
 * This entire class is just a skeleton for your code, plus a main method
 * for testing.
 */
public class GenericStack<T> {
        ArrayList<T> nodeData = new ArrayList<>();
        int size;


        public GenericStack(int capacity) {
            //nodeData = new String[capacity];
            nodeData = new ArrayList<>(capacity);
            size = 0;
        }

```

Develop mode **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → **GenericStack.java** (Your program) →

Program output displayed here

Coding trail of your work What is this?

```
10/12 W-0 R-0-7- min:20
```