COMP 250

Lecture 17

stack

Feb. 14, 2022

Recall: List operations

```
get(i)        //  Returns the i-th element (but doesn't remove it)
set(i,e)      //  Replaces the i-th element with e
add(i,e)      //  Inserts element e into the i-th position
remove(i)     //  Removes the i-th element from list
remove(e)     //  Removes first occurrence of element e
              //   from the list (if it is there)
clear()       //  Empties the list.
isEmpty()     //  Returns true if empty, false if not empty.
size()        //  Returns number of elements in the list
:
```

These operations can be defined abstractly, without specifying the implementation details (arraylist vs. linked list).

Abstract data type (ADT)

"ADT" defines a data type by the *values of the data* and *operations on the data.*

It is defined from the point of view of the *user*.

It ignores the details of the implementation.

An ADT is more abstract than a data structure.

It is *not* tied to a specific programming language.

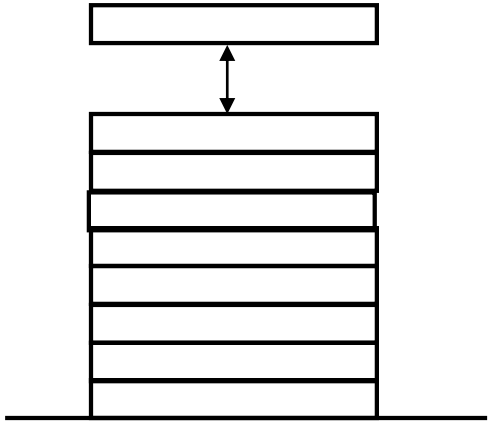
Stack ADT

push(element)

pop( )

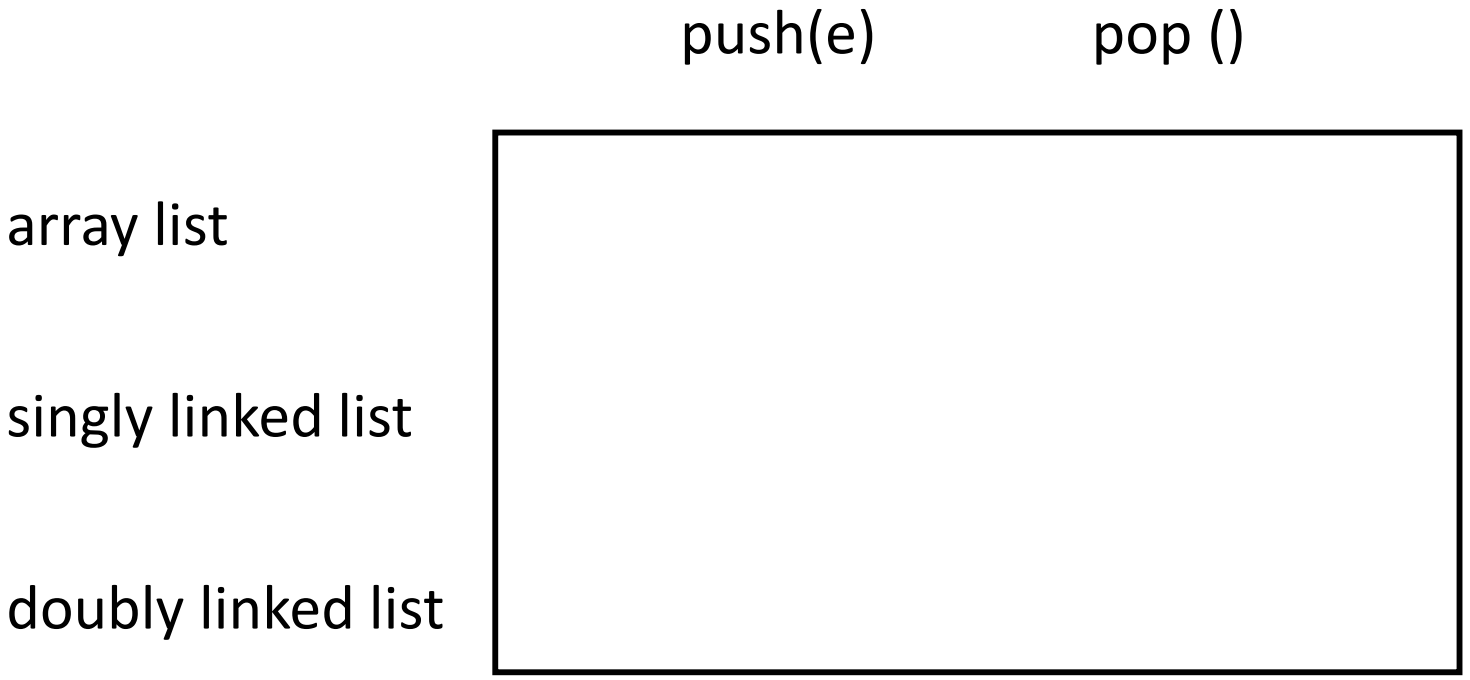
isEmpty( )

peek( )

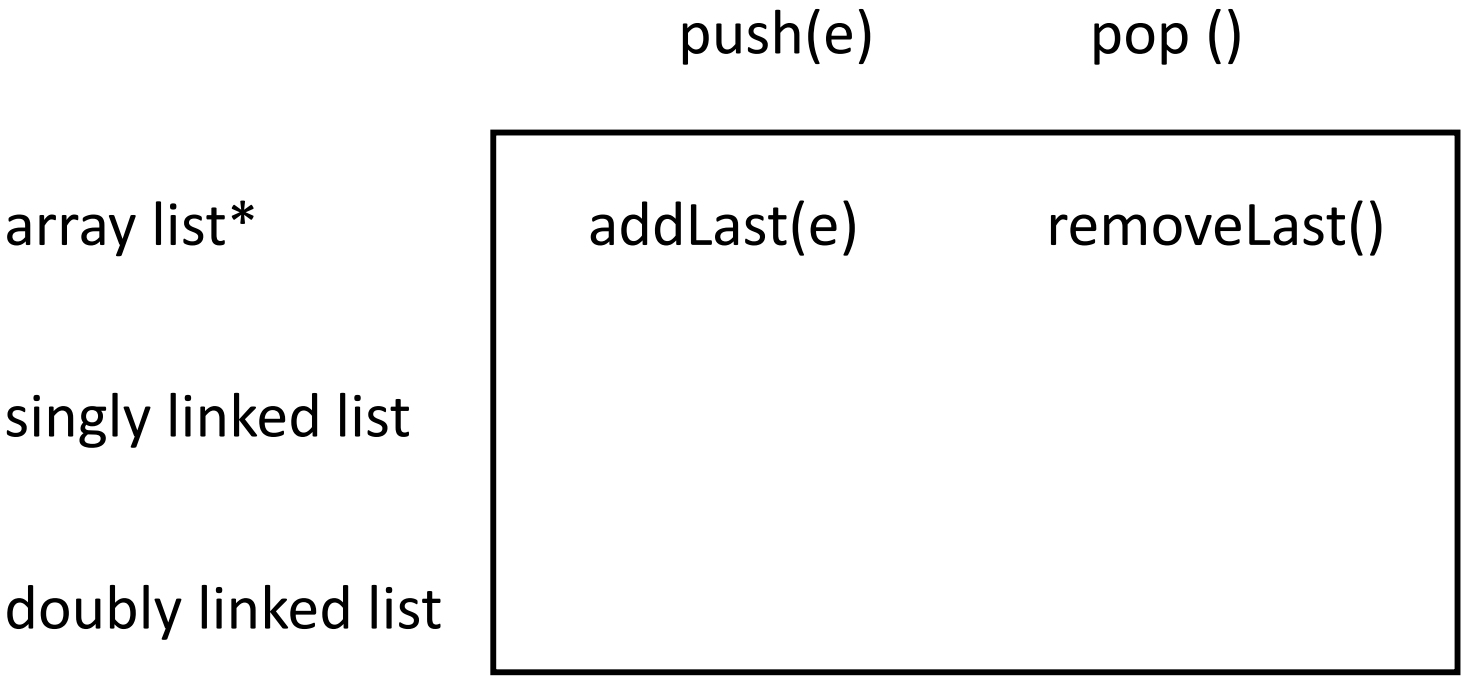


A stack is a list. However, it does not have operations to access the list element *i* directly. Instead one accesses only one end of the list.

How to implement a stack?



How to implement a stack?



*Java ArrayList class doesn't have addLast and removeLast methods. Use add(e) and remove(size()-1) instead.

How to implement a stack?

	push(e)	pop ()
array list	addLast(e)	removeLast()
singly linked list*	addFirst(e)	removeFirst ()
doubly linked list		

*Why not use addLast and removeLast with singly linked lists?

How to implement a stack?

	push(e)	pop ()
array list	addLast(e)	removeLast()
singly linked list	addFirst(e)	removeFirst ()
doubly linked list	either row above	

Example 1: stack of int

push(3), push(6), push(4), push(1), pop(), push(5), pop(), pop()….

Example 1: stack of int

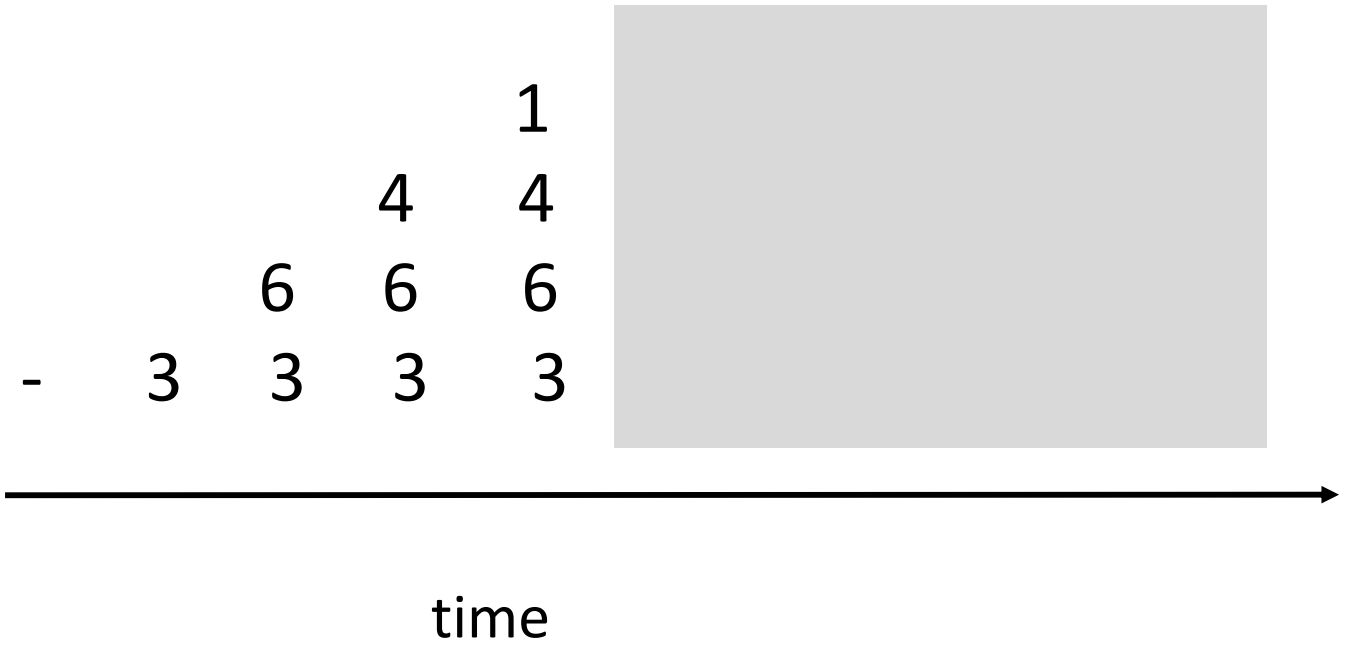
push(3), push(6), push(4), push(1), pop(), push(5), pop(), pop()….

Example 1: stack of int

push(3), push(6), push(4), push(1), pop(), push(5), pop(), pop()….

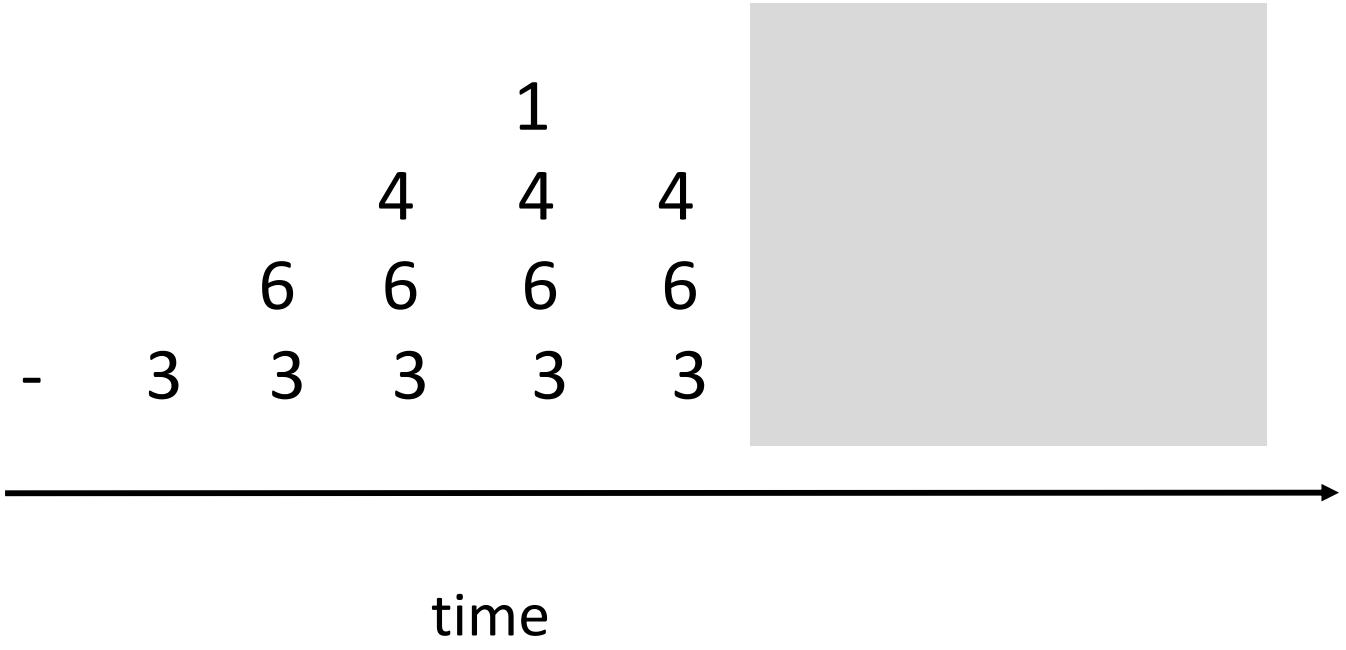
Example 1: stack of int

push(3), push(6), push(4), push(1), pop(), push(5), pop(), pop()….



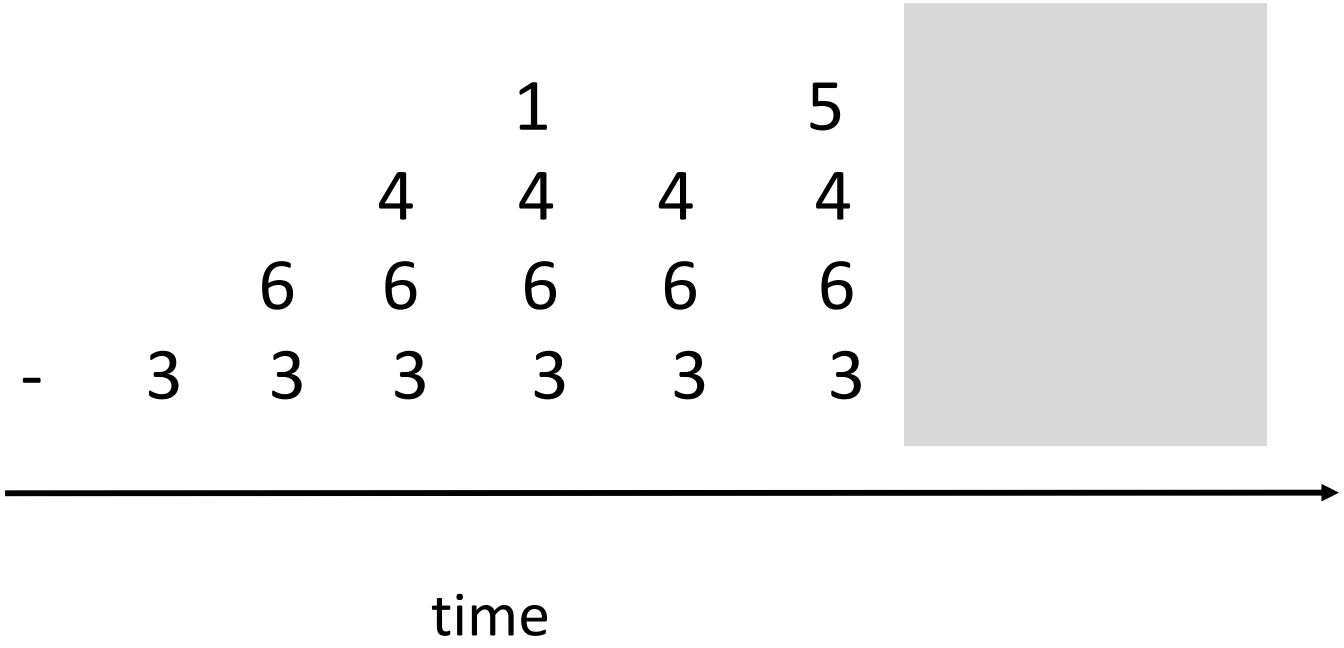
Example 1: stack of int

push(3), push(6), push(4), push(1), pop(), push(5), pop(), pop()….



Example 1: stack of int

push(3), push(6), push(4), push(1), pop(), push(5), pop(), pop()….



Example 1: stack of int

push(3), push(6), push(4), push(1), pop(), push(5), pop(), pop()….

Example 2 - balancing parentheses

e.g. (([ ])) [ ] { [ ] }

To ensure proper nesting, we traverse the list and use a stack.

How?

When we reach a *left* parenthesis, we *push* it onto the stack.

When we reach a *right* parenthesis, we compare it to top of the stack. If it matches, then we *pop*, else there is an error.

Example 2 - balancing parentheses

e.g. (([ ])) [ ] { [ ] }

Example 2 - balancing parentheses

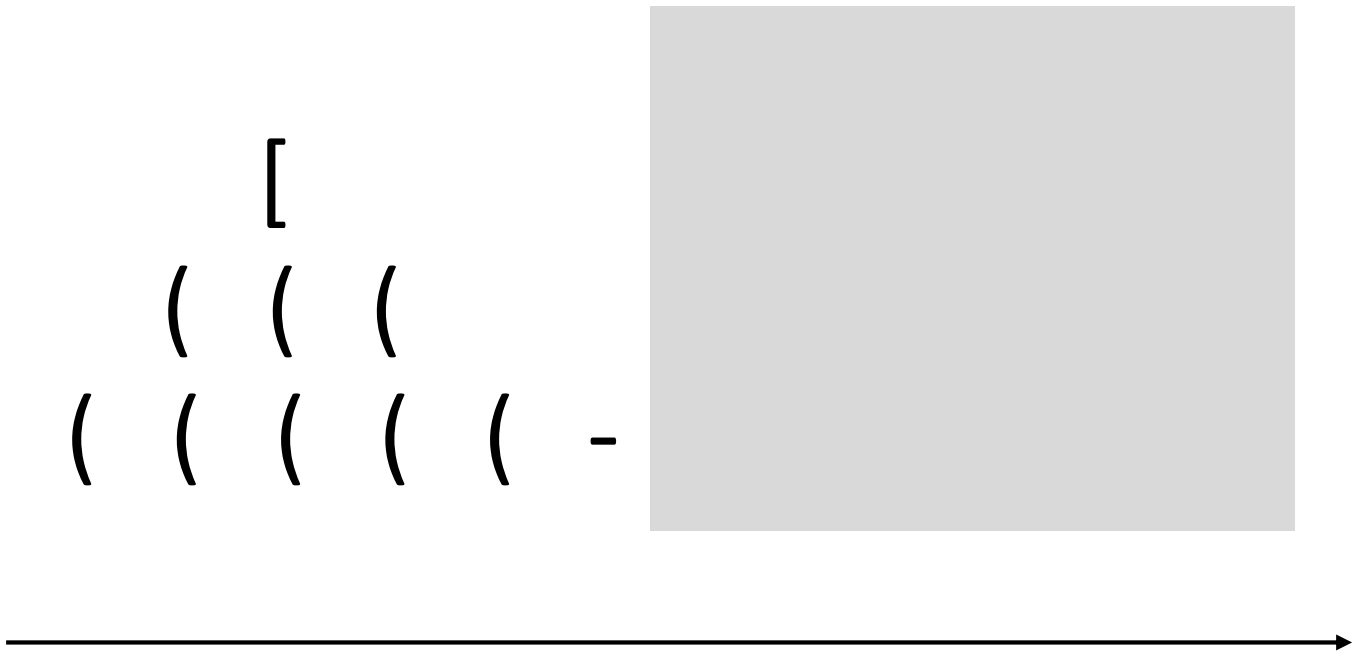
e.g. (([ ])) [ ] { [ ] }

Example 2 - balancing parentheses

e.g. (([ ])) [ ] { [ ] }

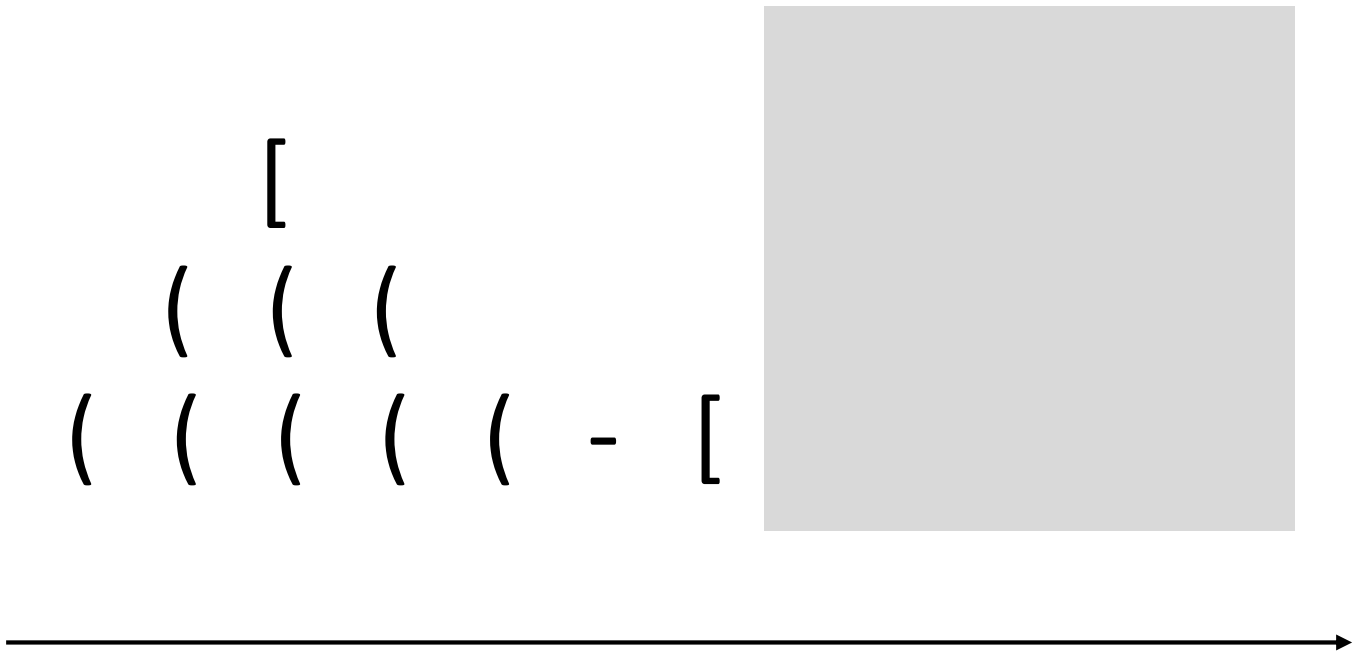
Example 2 - balancing parentheses

e.g. (([ ])) [ ] { [ ] }



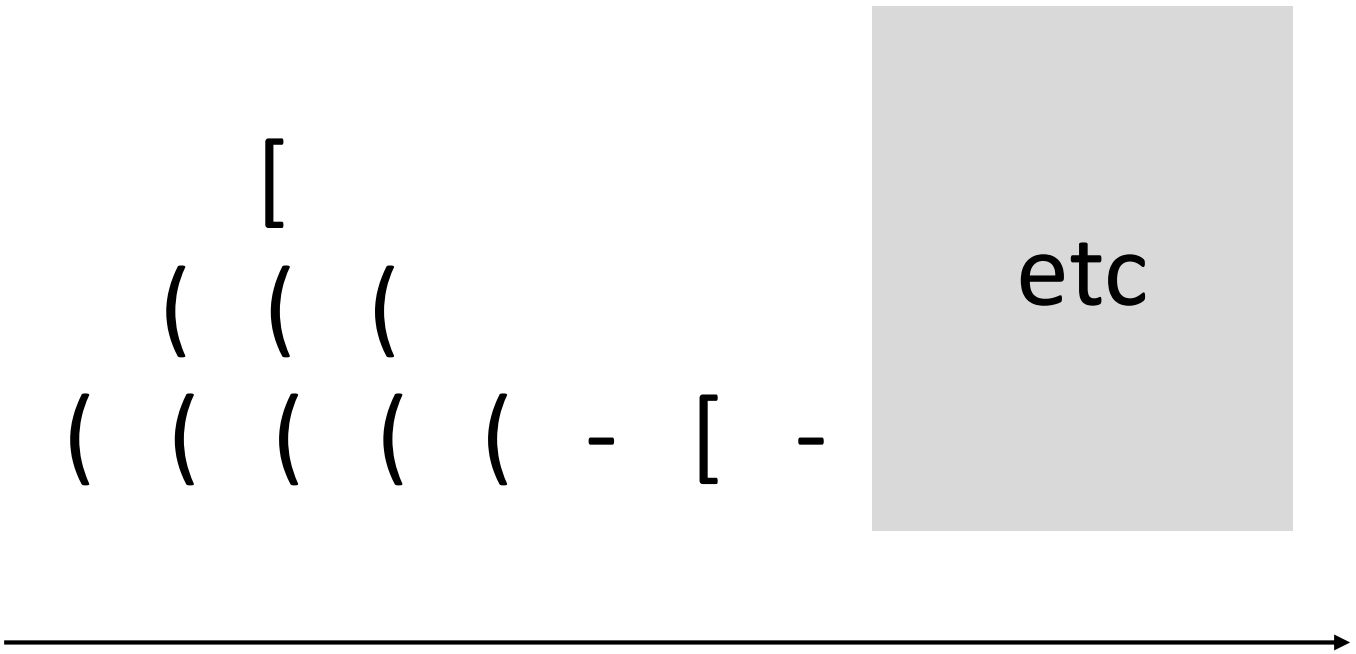
Example 2 - balancing parentheses

e.g. (([ ])) [ ] { [ ] }



Example 2 - balancing parentheses

e.g. (([ ])) [ ] { [ ] }



Example 2 - balancing parentheses

e.g. (([ ))] { [ ] }

Does not match left bracket on top of stack.

// We refer to brackets as “tokens”.
// This is the more general term used in *string parsing*.

Algorithm: decide if parentheses are matched (return true or false)

```
while (there are more tokens) {
    token = get next token
    if token is a left parenthesis
        push(token)
    else {          // token is a right parenthesis
            if stack is empty
                return false
            else {
                pop left parenthesis from stack
                if popped left parenthesis doesn't match the token (right parenthesis)
                    return false
                // else it does match, so continue
            }
        }
}
return stack.isEmpty().
```

Example 3: HTML tags

Suppose you want:

I am bold. *I am italic.*

In html, you would write:

 I am bold. < i > I am italic. < /i >

HTML Elements

An HTML *element* starts with a start tag.
An HTML *element* ends with an end tag.
These tags can be thought of as left and right brackets.

HTML documents consist of nested HTML *elements*.

```
<html>
<body>
<b>  I am bold </b>
<i>   I am italic </i>
</body>
</html>
```

Suppose you want:

I am bold. ***I am bold and italic.*** *I am italic.*

What if you were to write the following ?

 I am bold. I am bold and italic. I am italic.

This is *officially incorrect*, because elements are not nested.

< i >

____   Error: mismatch between

Most web browsers will interpret it correctly, however.

I am bold. ***I am bold and italic.*** *I am italic.*

The correct way to write it is:

 I am bold. I am bold and italic.   I am italic.

		< i >				
___	< b >	< b >	< b >	___	< i >	___

What problems can arise if you write it incorrectly?

Suppose you are editing a html document that contains the following:

Hello. I am bold.

<i> I am bold and italic. I am italic. < /i >

Bla bla bla ……

Q: What happens if you delete the middle line ?

What problems can arise if you write it incorrectly?

Suppose you are editing a html document that contains the following:

Hello. I am bold.

<i> I am bold and italic. I am italic. < /i >

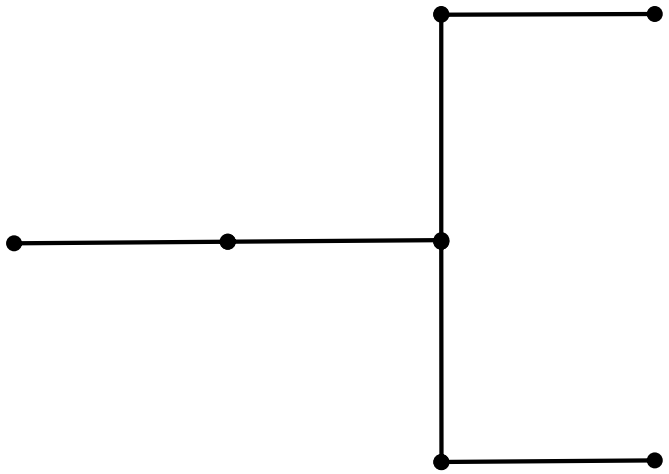
Bla bla bla ……

Q: What happens if you delete the middle line ?

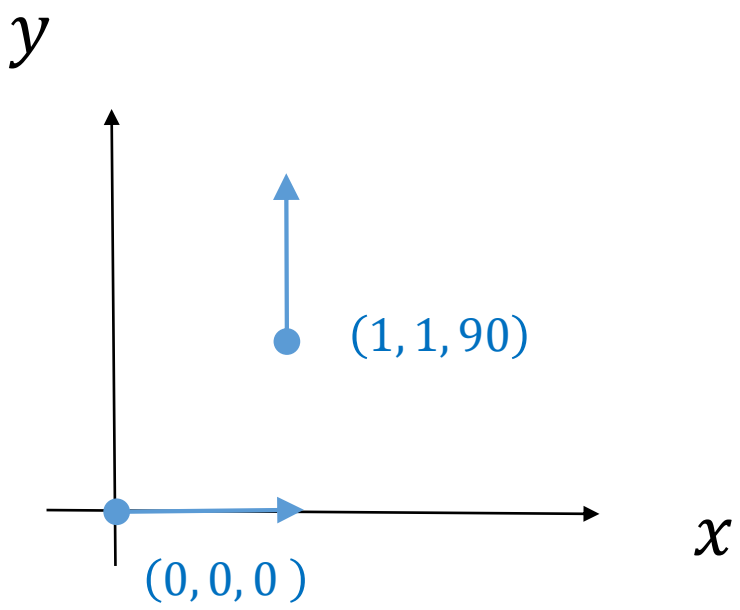
A: … Hello. **I am bold. Bla bla bla ……**

Example 4: Stacks in Graphics

Define a ‘programming language’ for drawing simple figures like this:



Define a pen position and direction (x, y, θ)
where θ is counter-clockwise degrees from x axis.



The initial *state of the pen* is $(0, 0, 0)$.
Later the state might be $(1, 1, 90)$

.

Let instructions be symbols :

D - draw unit length line in direction θ (changes (x, y))

R - turn right 90 degrees (changes θ)

L - turn left 90 degrees (changes θ)

[- push state (x, y, θ)

] - pop state, and go to that state

The initial state of the pen is $(x, y, \theta) = (0, 0, 0)$.

D D R D L D

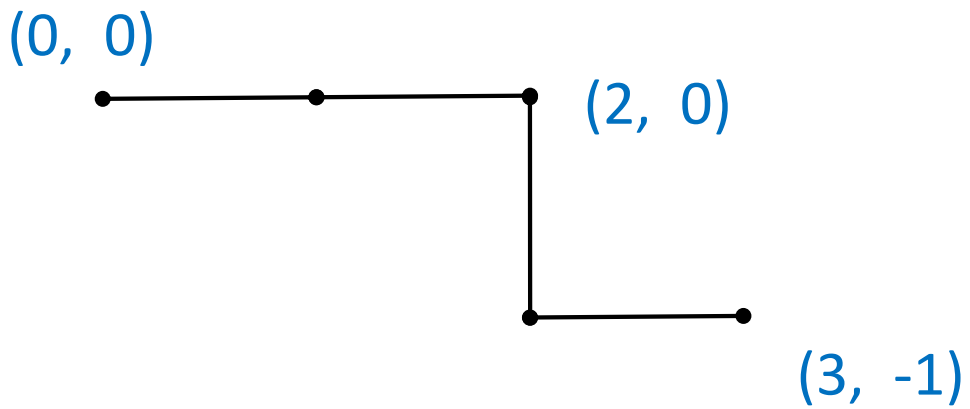
(0, 0)

D - draw
R - turn right 90 deg
L - turn left 90 deg
[- push state
] - pop state

The final pen state and drawn shape is ?

The initial state of the pen is $(x, y, \theta) = (0, 0, 0)$.

D D R D L D

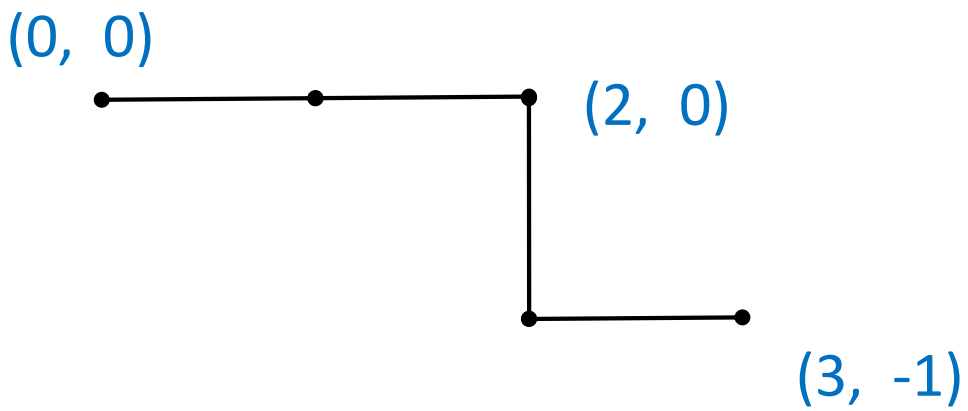


D - draw
R - turn right 90 deg
L - turn left 90 deg
[- push state
] - pop state

The final pen state is (3, -1, 0).

The initial state of the pen is $(x, y, \theta) = (0, 0, 0)$.

D D [R D L D]



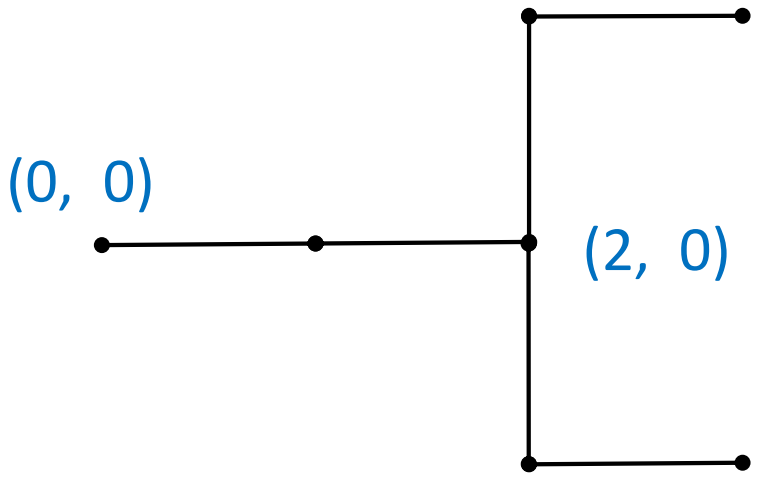
D - draw
R - turn right 90 deg
L - turn left 90 deg
[- push state
] - pop state

Q: What will be the final pen state ?

A: (2, 0, 0)

The initial state of the pen is $(x, y, \theta) = (0, 0, 0)$.

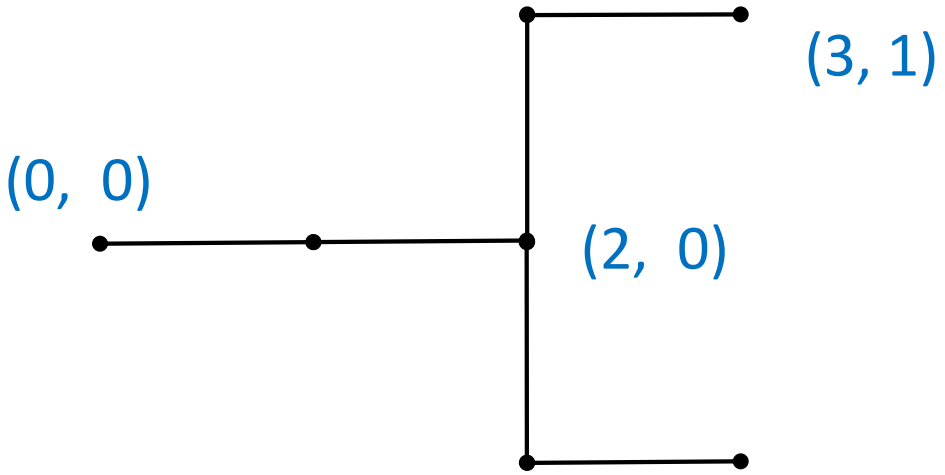
D D [R D L D] L D R D



Q: What will be the final pen state ?

A: (3, 1, 0)

The initial state of the pen is $(x, y, \theta) = (0, 0, 0)$.



Q: What if we add brackets at beginning and ending ?

[D D [R D L D] L D R D]

A: The pen state will return to (0, 0, 0).

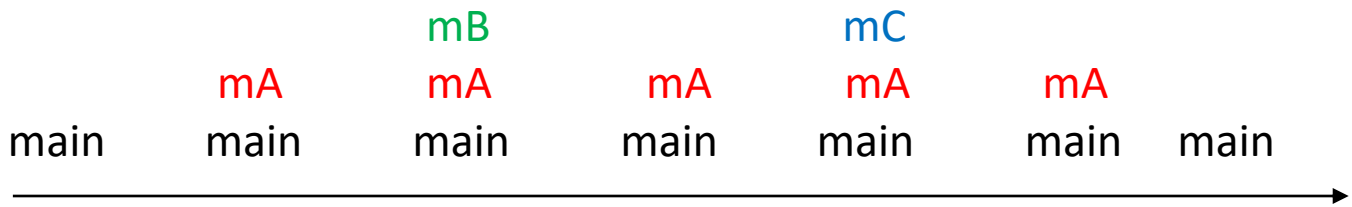
Example 5 : "Call Stack"

```
class   Demo {
    void   mA( ) {
               mB( );
               mC( );
           }
    void  mB( ) { … }
    void  mC( ) { … }

    public static void  main( ){
               mA(  );
       }
}
```

The call stack keeps track of which method was called by which method (which was called by which method, ...)

How does the call stack evolve over time as the program executes ?

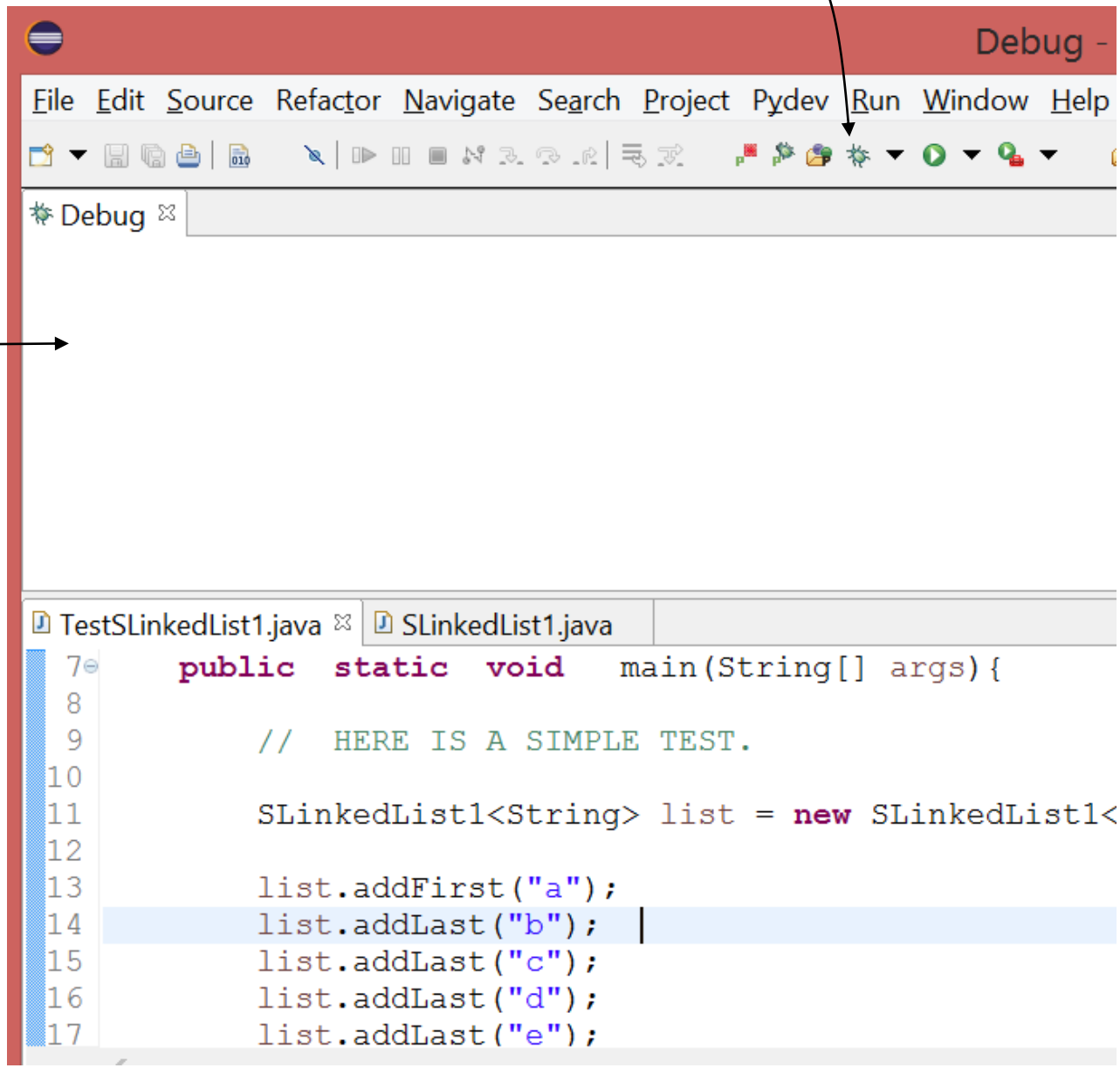
```
class    Demo {
     void   mA( ) {
                mB( );
                mC( );
            }
     void  mB( ) { … }
     void  mC( ) { … }

     public static void  main( ){
                mA(  );
        }
}
```



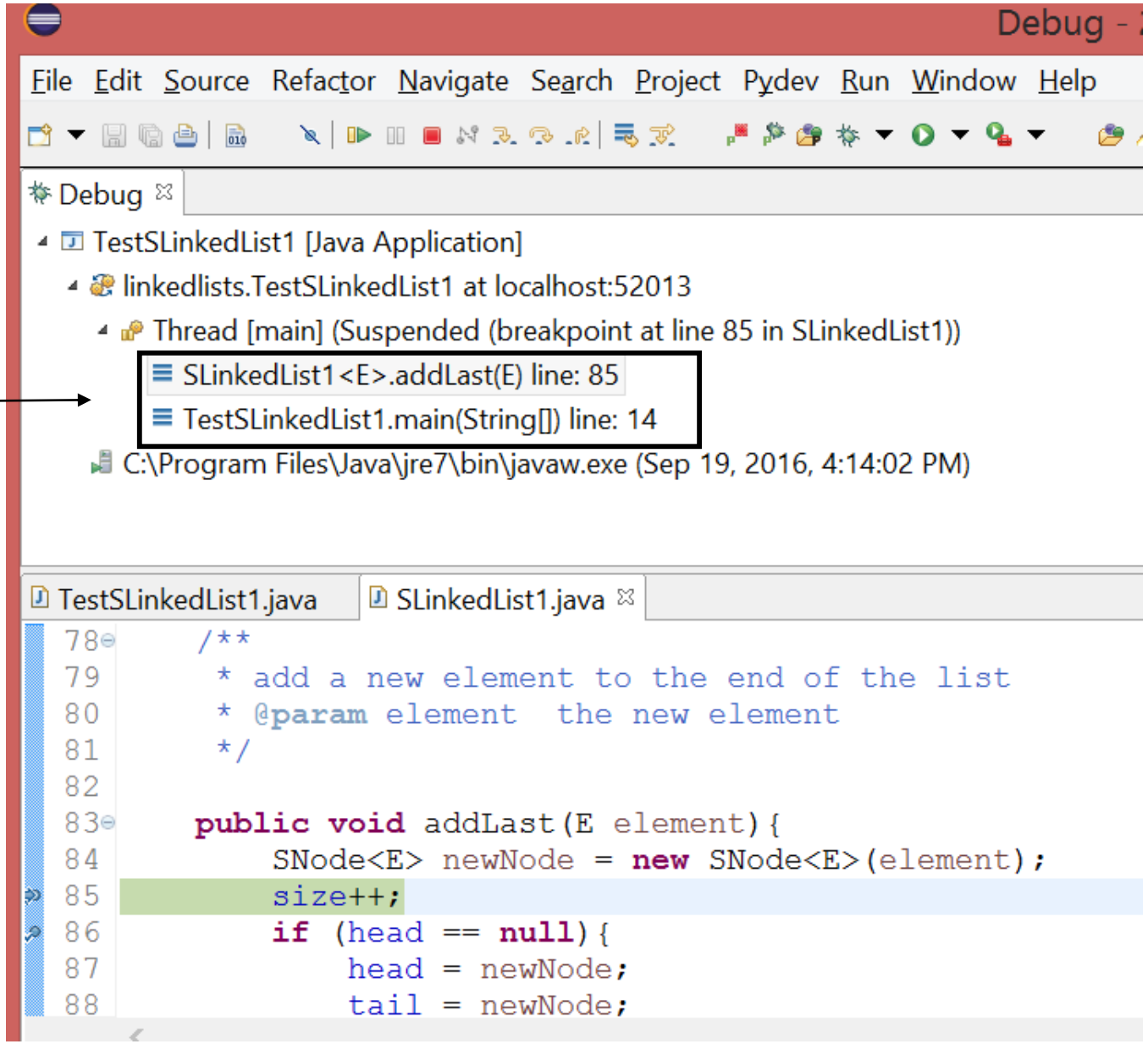
Eclipse debug mode

call stack
will appear here

TestSLinkedList1.main()
calls
SLinkedList.addLast()



call stack



Breakpoint within the SLinkedList1. addLast() method

“Stack overflow”

If a stack has a finite capacity, and the stack reaches that capacity, and we attempt to push again …

“Stack underflow”

If a stack is empty, and we pop…

Coming up…

Lectures

Wed. Feb. 16 Queues

Fri. Feb. 18 Mathematical Induction

Next week... recursion

Assessments

Assignment 1 grades will be released later today along with the grader code.

Assignment 2 is due on Fri. Feb. 25.

Assignment 3 will be released shortly after that. (READING WEEK)