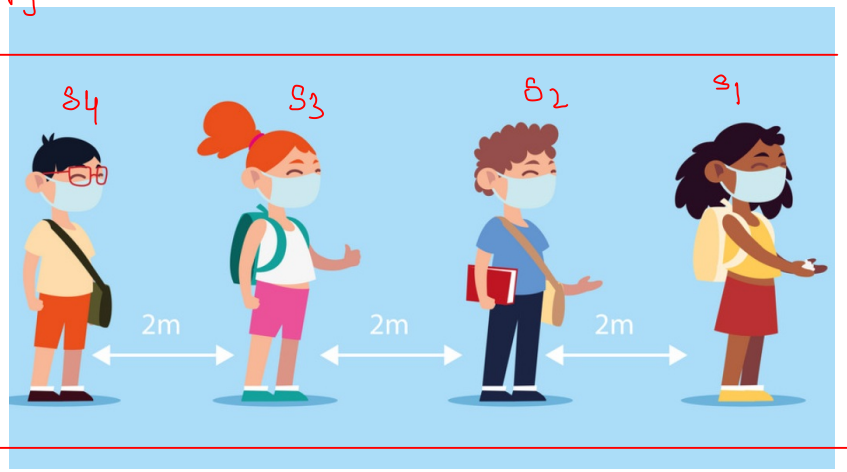
S3-Class1 [Stack-1]

student



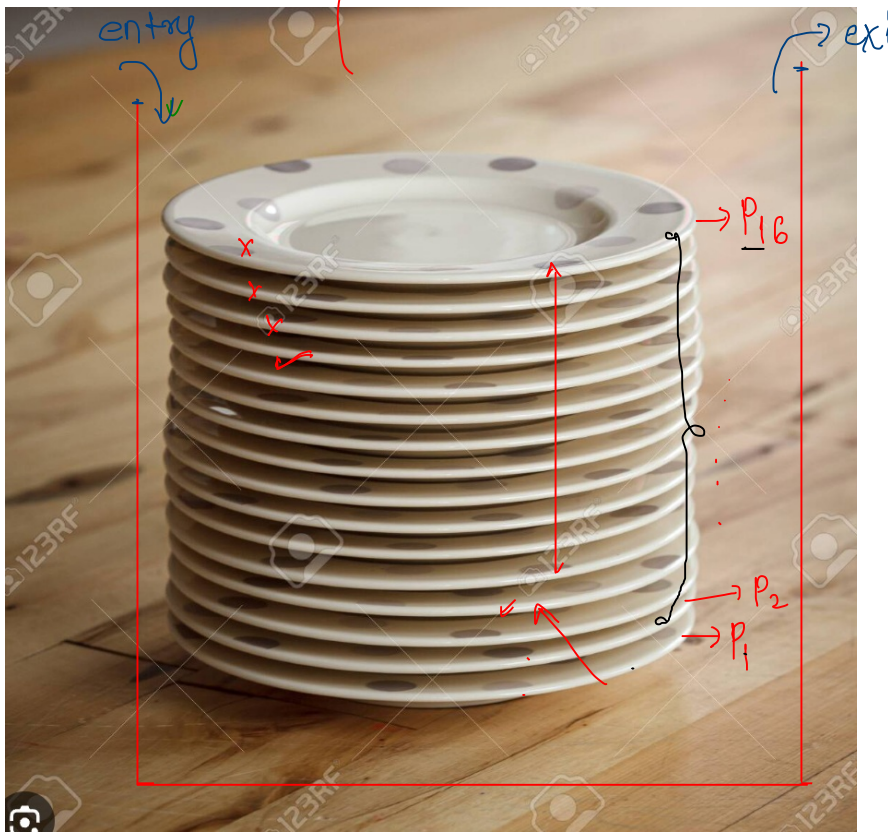
Queue

↳ First In First out (FIFO)

s_1 : comeout first

2 sides open { one side → entry
other side → exit

stack of plates



P_{16}

Last In first out

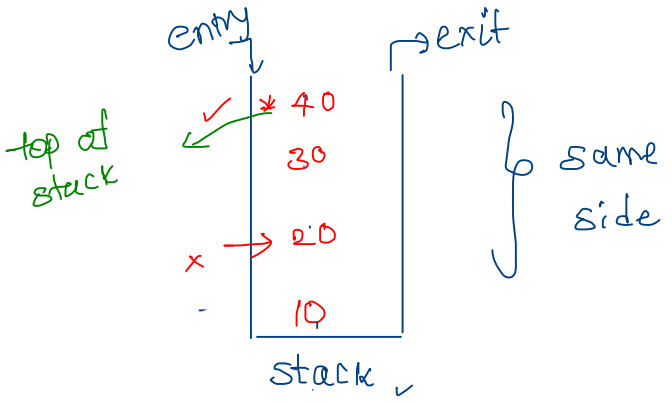
(LIFO)

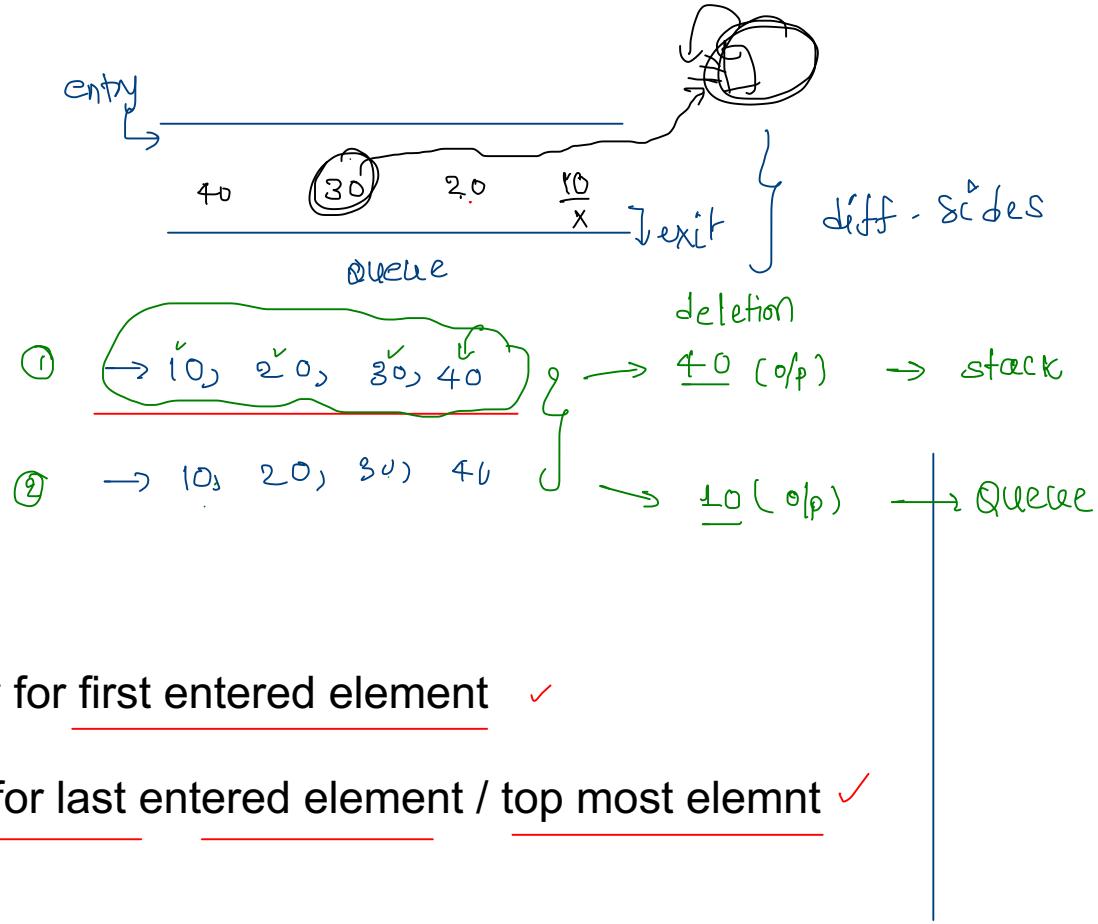
First In Last out ✓

(FILO)

Mess worker

Last plate P_{16}





FIFO

1) in Queue : Access will be there only for first entered element ✓

2) in Stack : Access will be there only for last entered element / top most elemnt ✓

↳ LIFO

↳ int, float, long, etc. . . .

↳ primitive Data Type / system Defined D.T / inbuilt

↳ New D.T (ADT)

array:-

↳ same Datatype

↳ contains new locations

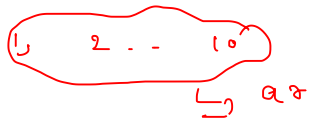
int arr[] = new int[10]

String arr[] = new String[ ]

abstract :- existing in thought or as an idea but not having a physical or concrete existance

DS

Xyz



Stack:-

-> Stack is an " abstract data type [ADT] ", that serves as a collection of elements with two main operations

1. push : which adds an element to the collection
2. pop : which removes the most recently added element that was not yet removed

-> The order in which an element added to or removed from a stack is described as

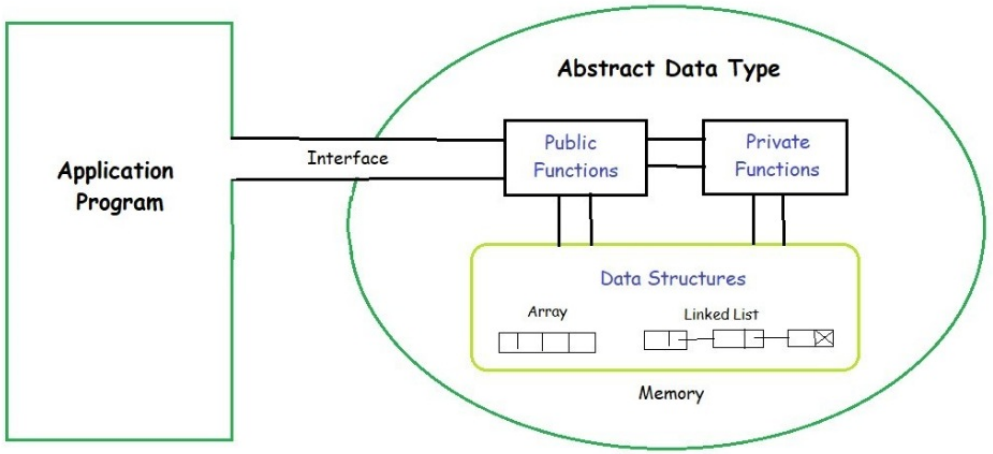
Last In First Out [LIFO]

First In Last Out [FILO]

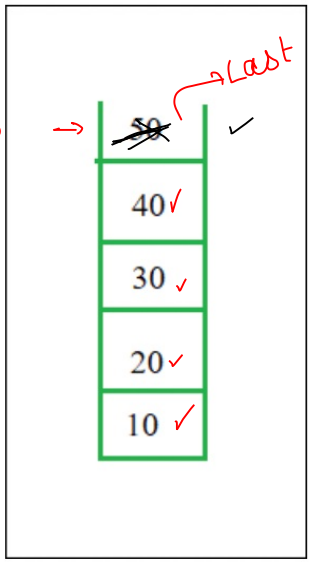
In this article, we will learn about ADT but before understanding what ADT is let us consider different in-built data types that are provided to us. Data types such as int, float, double, long, etc. are considered to be in-built data types and we can perform basic operations with them such as addition, subtraction, division, multiplication, etc. Now there might be a situation when we need operations for our user-defined data type which have to be defined. These operations can be defined only as and when we require them. So, in order to simplify the process of solving problems, we can create data structures along with their operations, and such data structures that are not in-built are known as Abstract Data Type (ADT).

Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of values and a set of operations. The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called “abstract” because it gives an implementation-independent view.

The process of providing only the essentials and hiding the details is known as abstraction.



2. Stack ADT

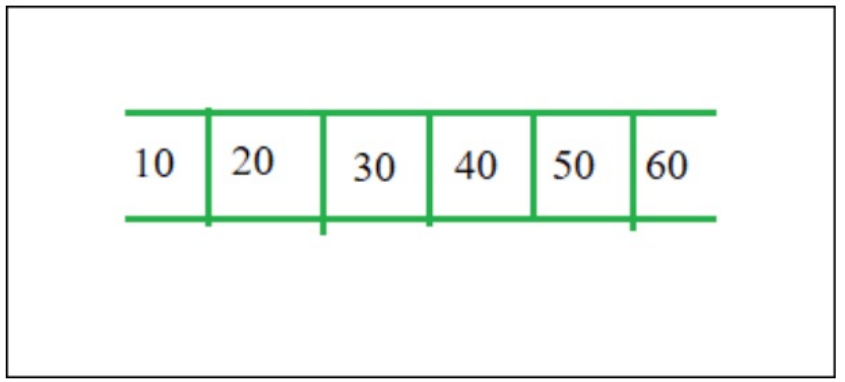


View of stack

- ~~In Stack ADT Implementation instead of data being stored in each node, the pointer to data is stored~~.
- The program allocates memory for the *data* and *address* is passed to the stack ADT.
- The head node and the data nodes are encapsulated in the ADT. The calling function can only see the pointer to the stack.
- The stack head structure also contains a pointer to *top* and *count* of number of entries currently in stack.

1) • push() – Insert an element at one end of the stack called top.
2) • pop() – Remove and return the element at the top of the stack, if it is not empty.
3) • peek() – Return the element at the top of the stack without removing it, if the stack is not empty.
4) • size() – Return the number of elements in the stack.
5) • isEmpty() – Return true if the stack is empty, otherwise return false.
6) • isFull() – Return true if the stack is full, otherwise return false.

3. Queue ADT



View of Queue

- The queue abstract data type (ADT) follows the basic design of the stack abstract data type.
- Each node contains a void pointer to the *data* and the *link pointer* to the next element in the queue. The program's responsibility is to allocate memory for storing the data.
- enqueue() – Insert an element at the end of the queue.
- dequeue() – Remove and return the first element of the queue, if the queue is not empty.
- peek() – Return the element of the queue without removing it, if the queue is not empty.
- size() – Return the number of elements in the queue.
- isEmpty() – Return true if the queue is empty, otherwise return false.
- isFull() – Return true if the queue is full, otherwise return false.

Consider a stack with size of 5 elements, Assume initially stack is empty and apply following operations on stack

1) push(10)

2) push(20)

3) push(30)

4) pop() → 30

5) push(40) →

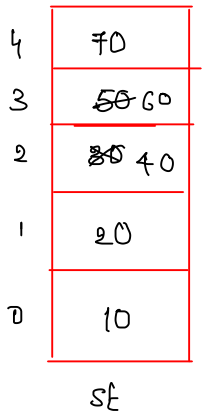
6) push(50) ✓

7) pop() → 50

8) push(60) →

9) push(70) →

10) push(80) → isFull ⇒ overflow error



	Adding ele	Deleting ele
Stack	push(x)	pop()
Queue	enqueue(x)	dequeue()

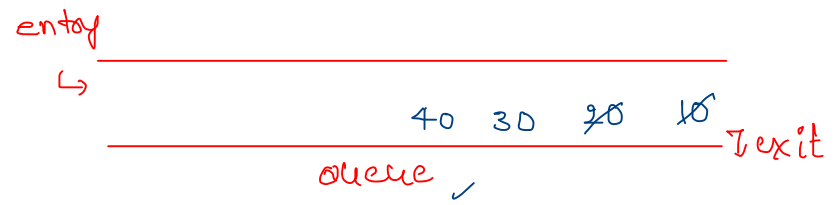
} Standard Names

May change, when it comes to prog lang to prog lang.

Queue():-

E(10), E(20), E(30), DQ(), E(40), DQ()
↳ 10 ↳ 20

1) FIFO



2) Enqueue() → adds ele to Queue.

3) Dequeue() → delete ele from Queue

UGC NET CSE | January 2017 | Part 2 | Question: 22

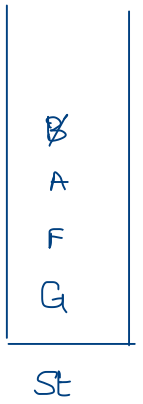
asked in DS Mar 24, 2020 • recategorized May 24, 2020

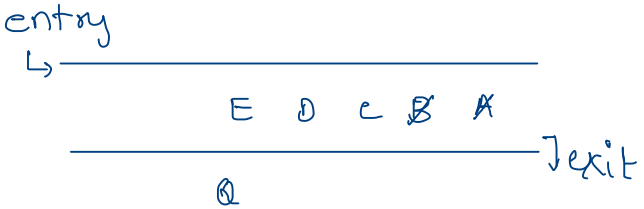
6,079 views

1

The seven elements A, B, C, D, E, F and G are pushed onto a stack in reverse order, i.e., starting from G. The stack is popped five times and each element is inserted into a queue. Two elements are deleted from the queue and pushed back onto the stack. Now, one element is popped from the stack. The popped item is _____.

A. A
B. B
C. F
D. G





GATE CSE 2023 | Question: 49

asked in DS Feb 15 • edited Mar 19 by **Lakshman Bhaiya**

2,031 views

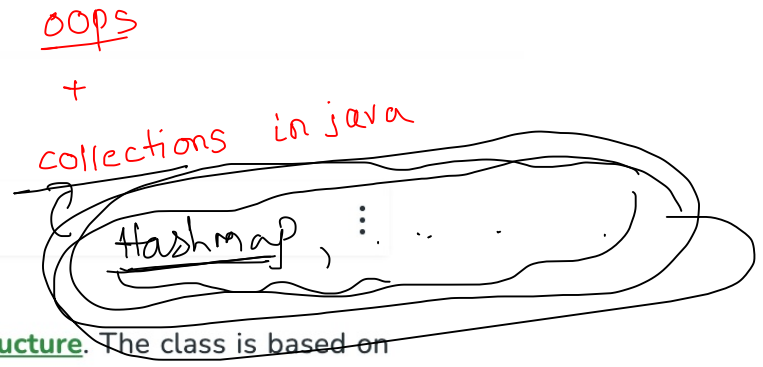
3

Consider a sequence a of elements $a_0 = 1, a_1 = 5, a_2 = 7, a_3 = 8, a_4 = 9$, and $a_5 = 2$. The following operations are performed on a stack S and a queue Q, both of which are initially empty.

I. **push** the elements of a from a_0 to a_5 in that order into S.
II. **enqueue** the elements of a from a_0 to a_5 in that order into Q.
III. **pop** an element from S.
IV. **dequeue** an element from Q.
V. **pop** an element from S.
VI. **dequeue** an element from Q.
VII. **dequeue** an element from Q and push the same element into S.
VIII. Repeat operation VII three times.
IX. **pop** an element from S.
X. **pop** an element from S.

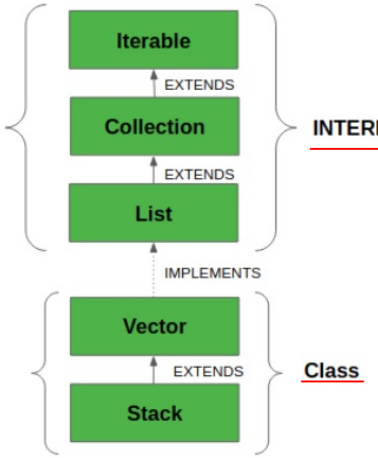
The top element of S after executing the above operations is _____8 ✓_____.

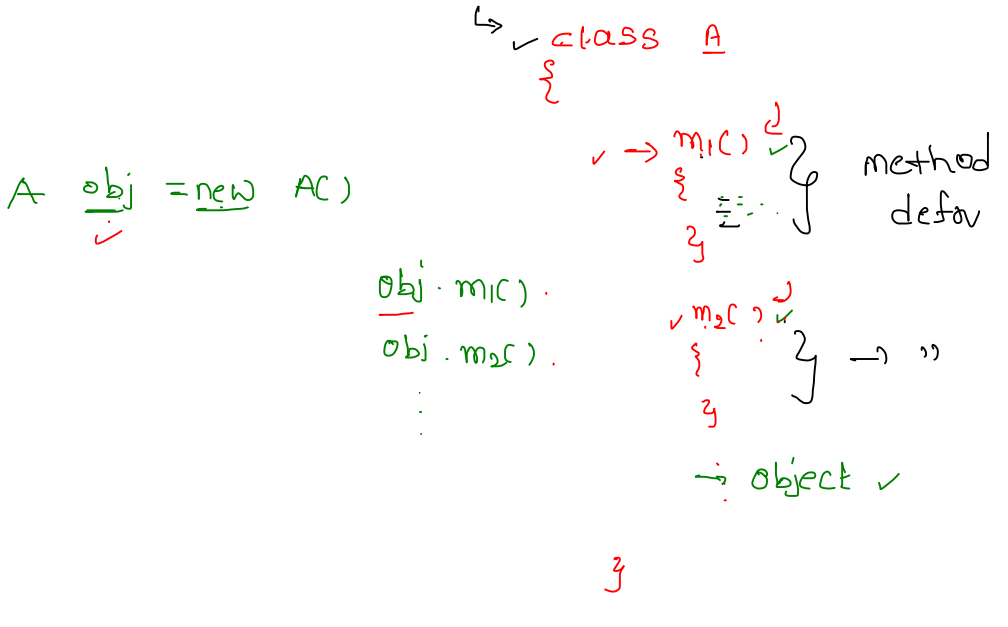
Stack Class in Java



Java Collection framework provides a Stack class that models and implements a **Stack data structure**. The class is based on the basic principle of last-in-first-out. In addition to the basic push and pop operations, the class provides three more functions of empty, search, and peek. The class can also be said to extend Vector and treats the class as a stack with the five mentioned functions. The class can also be referred to as the subclass of Vector.

The below diagram shows the **hierarchy of the Stack class**:





* * *

Hashmap

Key | Value

1) How to declare ✓

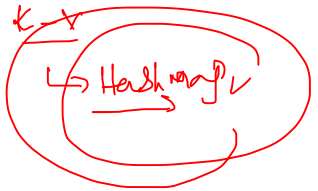
2) How to add element ✓

3) How to delete element ✓

4) How to access element

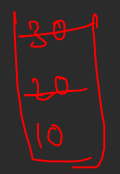
5) How to find size ✓

```java
package sprint3;

import java.util.Stack;

public class First {
    public static void main(String[] args) {
        Stack<Integer> st=new Stack<>();
        st.push( item: 10);
        st.push( item: 20);
        st.push( item: 30);
        System.out.println(st.peek());

    }
}
```



```
First
/Library/Java/JavaVirtualMachines/jdk-17.0.4.jdk/Contents/Home/bin/java -javaage
30

Process finished with exit code 0
```

```java
package sprint3;

import java.util.Stack;

public class First {
    public static void main(String[] args) {
        Stack<Integer> st=new Stack<>();
        st.push( item: 10);
        st.push( item: 20);
        st.push( item: 30);
        st.pop();
        st.pop();
        System.out.println(st.peek());

    }
}
```



↳ 10

First

```
/Library/Java/JavaVirtualMachines/jdk-17.0.4.jdk/Contents/Home/bin/j
10
```

Main.java | SubsetsHavingSumK.java | First.java | Nick.java | Sample.java

```java
package sprint3;

import java.util.Stack;

public class First {
    public static void main(String[] args) {
        Stack<Integer> st=new Stack<>();
        st.push( item: 10);
        st.push( item: 20);
        st.push( item: 30);
        st.pop();
        st.pop();
        System.out.println(st.peek());
        st.pop();
        System.out.println(st.peek());

    }
}
```

10

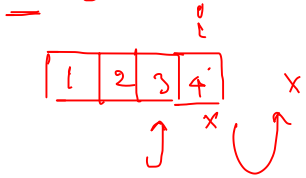
↳ exception

First

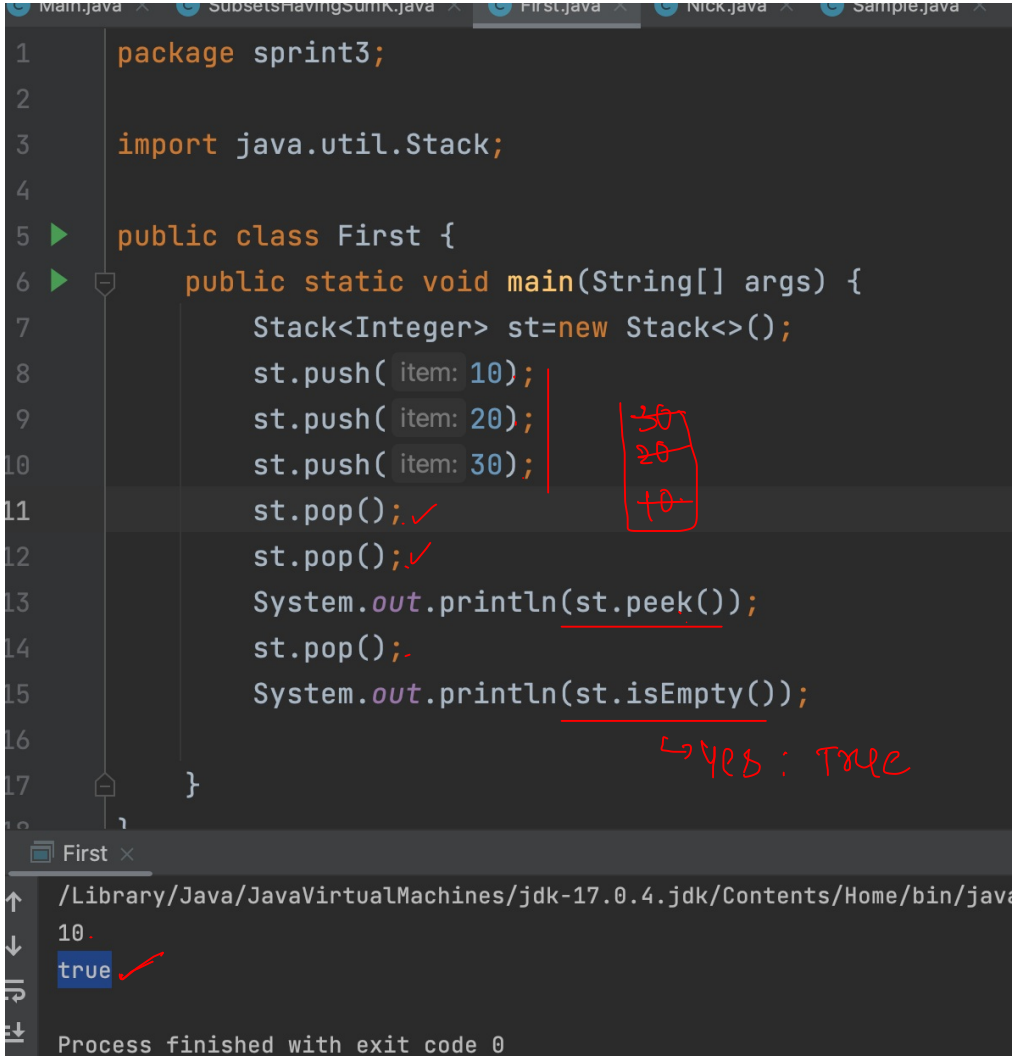
```
/Library/Java/JavaVirtualMachines/jdk-17.0.4.jdk/Contents/Home/bin/java -javaa
10
Exception in thread "main" java.util.EmptyStackException Create breakpoint
    at java.base/java.util.Stack.peek(Stack.java:101)
    at sprint3.First.main(First.java:15)

Process finished with exit code 1
```

① AIOBE

a[i], a[i+1]



② ESException



```java
package sprint3;

import java.util.Stack;

public class First {
    public static void main(String[] args) {
        Stack<Integer> st=new Stack<>();
        st.push( item: 10);
        st.push( item: 20);
        st.push( item: 30);
        st.pop();
        st.pop();
        System.out.println(st.peek());
        st.pop();
        System.out.println(st.isEmpty());

    }
}
```

```
First
/Library/Java/JavaVirtualMachines/jdk-17.0.4.jdk/Contents/Home/bin/java
10
true

Process finished with exit code 0
```

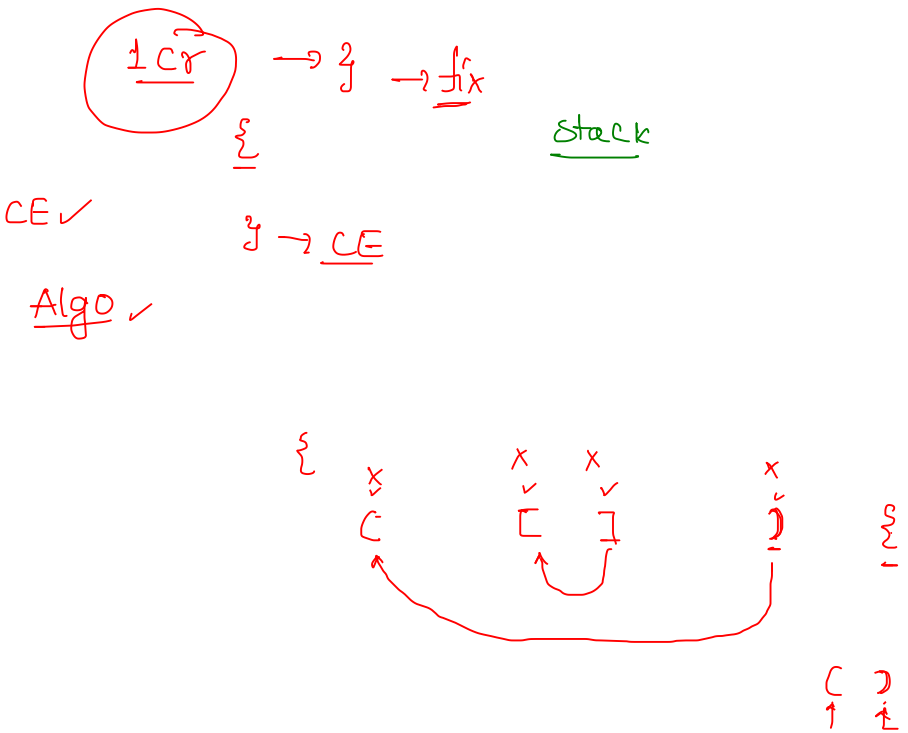
12:40pm

```java
package sprint3;

import java.util.Stack;

public class First {
    public static void main(String[] args) {
        Stack<Integer> st=new Stack<>();
        st.push( item: 10);
        st.push( item: 20);
        st.push( item: 30);
        st.pop();
        st.pop();
        System.out.println(st.peek());
        st.pop();
        System.out.println(st.isEmpty());


    }
}
```

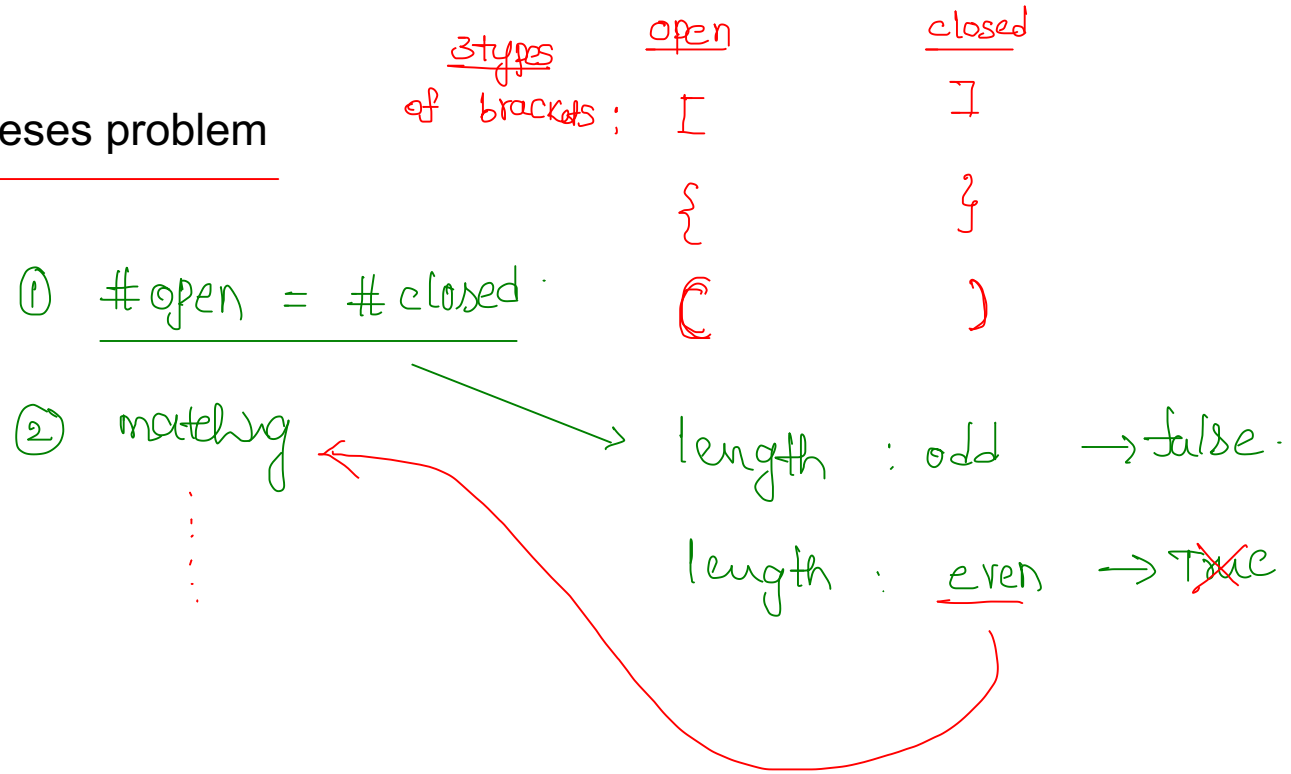
50,000 lines
1 line
} → CE ✓

First

```
/Library/Java/JavaVirtualMachines/jdk-17.0.4.jdk/Contents/Home/bin/java
10
true

Process finished with exit code 0
```



1) Balanced Parantheses problem



in a expression as long as open brackes there is no problem,
when you see the first closing bracket, then it should match with recent opened bracket

the given expression MUST satisfy the above two conditions

exp1 :

{ [( ) }]

stop

( )

[ ]

{ }

open → no problem

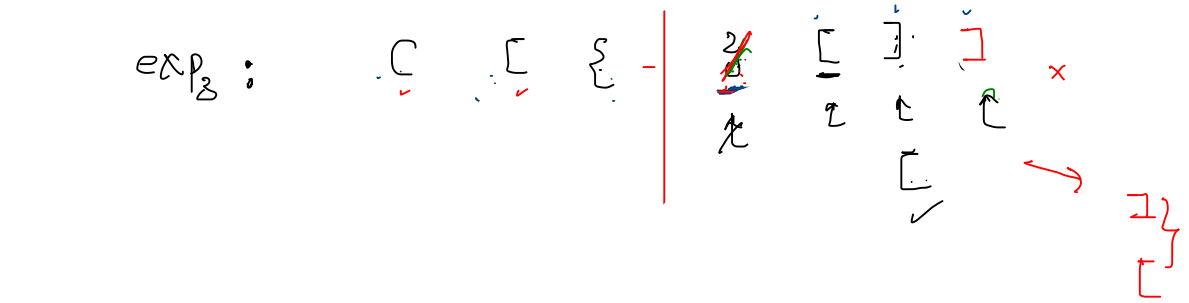
exp2 :-

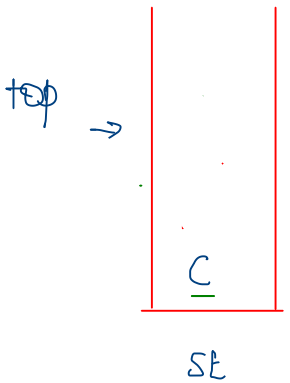
[{ ( ) [ ] { }) }]

[{) }]

stop

exp3 :





→ stack is Empty!

↳ more open brackets than closed

```
boolean isBalanced(char arr[], int n)
{
	if(n%2==1) return false;
	Stack<Character>st=new Stack<>();
	for(int i=0;i<n;i++)
	{
		char ele=arr[i];
		if(ele=='(' || ele=='[' || ele=='{') // push to stack
		{
			st.push(ele);
		}
		else  // if input is closed bracket, pop from stack
		{
			if(st.isEmpty())
				return false;
			char out=st.pop()
			if(ele==')' && out!='(')
				return false;
			if(ele==']' && out!='[')
				return false;
			if(ele=='}' && out!='{')
				return false;
		}
	}
	return st.isEmpty()
}
```