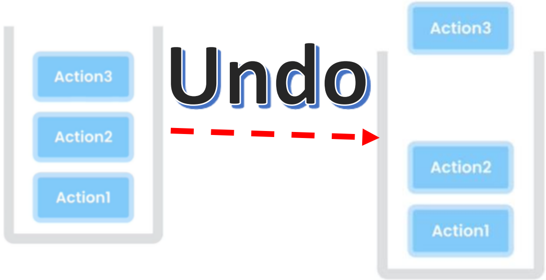
Stacks

- Structure of Stacks
- Runtime Complexity
- Solving real problems
- Build a Stack

(V-1) What Are Stacks?

STACKS

- Implement the undo feature
- Build compilers (eg syntax checking)
- Evaluate expressions (eg 1 + 2 * 3)
- Build navigation (eg forward/back)



The Last Action Can Be undone using Stacks

OPERATIONS

- push(item)
- pop()
- peek()
- isEmpty()

See The List of operations carefully we don't have look up here. That means we don't use stacks for store any list of customers, products . For lookup we use Array and Linked List .

All Operations has a time complexity of O(1) no matter it is implemented using Array or List.

(V-3)-Working with stacks

```
Stack <Integer> stack = new Stack<>();
stack.push(10);// [10] -> top -> 10
stack.push(20);// [10,20] -> top -> 20
stack.push(30);// [10,20,30] -> top -> 30
System.out.println(stack);
var pop = stack.pop();// pop store 30 as 30 is in the top of the stack
System.out.println(pop);
System.out.println(stack);// [10, 20] -> top -> 20
stack.pop(); // 20 will be popped as 20 is in the top
System.out.println(stack.peek()); // it will return the top value without removing
any items form stack. So, it will print 10
```

(V-4 & 5)-Reversing a String:-

```
import java.util.Stack;

public class StringReverser {
    // We can use stack DS to reverse a String Cause it is LIFO Structured
    public String reverse(String input){
        if (input==null)
            throw new IllegalArgumentException();
        Stack<Character> stack2 = new Stack<>();
        /* One Way using traditional For loop
        for(int i=0;i<input.length();i++)
        stack2.push(input.charAt(i));
        * */
        //but for each loop is better and tricky
        for(char ch : input.toCharArray()){//input.toCharArray() is written cause in
java for each loop we can't iterate through String thats why first we make it in
chararry
            stack2.push(ch);
        }
        // Now half of the operation is done just left popping the elements from stacks
        // But if we declare a String var and everytime concatanating it with new
characters
        // it will become a more time consuming cause in JAVA String is immutable
        // That means we can't directly manipulate the String using same memory
        // Every time String changes new memory will be allocated for the same
operations
        // So we can use StringBuffer class . It is best for String Manipulation when
too much string manipulation is needed
        StringBuffer reversed = new StringBuffer();
        while(!stack2.empty())// all characters will be popped and string will be
reversed
            reversed.append(stack2.pop());// all charactes popped doing like
concatanation but in same memory. Memory is saved now.
        return reversed.toString();// charecter converted to String and then returned
    }
}
```

V-(6-10)-(Balance Parenthesis):-[All Basic Operation's and uses of stacks are covered in this part]

Mosh Solution 1:-

```
public boolean isBalanced(String input){
   // The idea is for checking the parenthesis are balanced or not using undo operation
   // We iterate through the whole String (Character's) and push if we found opening bracket
   // if we found closing bracket we then pop
   // if the stack is empty then all parenthesis are balanced
   // if we have anything left on the stack that means expression is not balanced
   Stack <Character> stack = new Stack<>();// Call Stack
   for(char ch : input.toCharArray()){// for each loop we can't iterate through string so we
convert it into characters
       if(ch == '(' || ch == '{' || ch == '[' || ch == '<')
          stack.push(ch);
       if(ch == ')' || ch == '}' || ch == ']' || ch == '>' ){
          if(stack.empty()) return false;
          var top = stack.pop();
          if (// As we have total 4 types of bracket so we pop for every closing bracket and
check with the corresponding current right bracket if they don't match as per required we
return false immediately
                      (ch == ')' && top!= '(') ||
                      (ch == '}' && top!='{') ||
                      (ch == ']' && top!='[') ||
                      (ch == '>' && top!='<')
          ) return false;
       }
   }
   return stack.empty();
}
```

Mosh Solution 2(More Readable Refactored):-

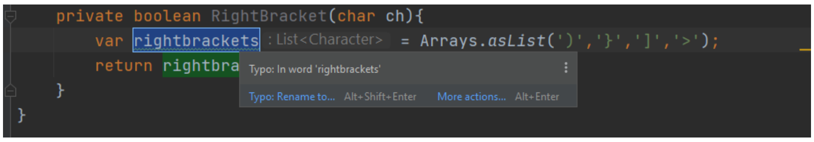
```
public boolean isBalanced2(String input1){
   Stack<Character> stack1 = new Stack<>();
   for(char ch : input1.toCharArray()){//pura string charcter array te niye iterate
   if(isLeftBracket(ch))// left bracket (,{,[,< paile push
      stack1.push(ch);
   if(isRightBracket(ch)) {// right bracket ),},],> paile pop kore compare
     if(stack1.empty()) return false;
     var top = stack1.pop();
     if((BracketMatch(top,ch))) return false; //eikhane BracketMatch false
   }
   }
   return stack1.empty();
}
```

```
private boolean isLeftBracket(char ch){// opening bracket gula thakle push korbe
//call hoya method theke
    return ch == '(' || ch == '{' || ch == '[' || ch == '<';
}
private boolean isRightBracket(char ch){
    return ch == ')' || ch == '}' || ch == ']' || ch == '>';
}
private boolean BracketMatch(char left,char right){//pop korbe opening bracket
// (,[,{,< aigula left bracket tai namkoron e char left dicci
// ),],},> aigula right(closing) bracket jeigula stack e push hobe nah but compare
//hobe tai jeikhan theke call hocce seikhan theke current character patano hoyese
   return  (right == ')' && left!= '(') ||
           (right == '}' && left!='{') ||
           (right == ']' && left!='[') ||
           (right == '>' && left!='<');
}
```

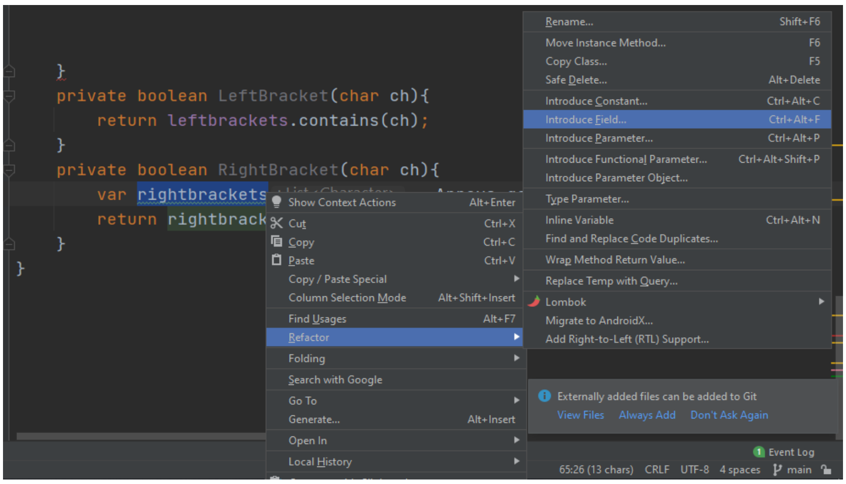
Mosh Solution 3(More Readable Refactored+):-

Firstly, We need to know about Extracting fields form any method:

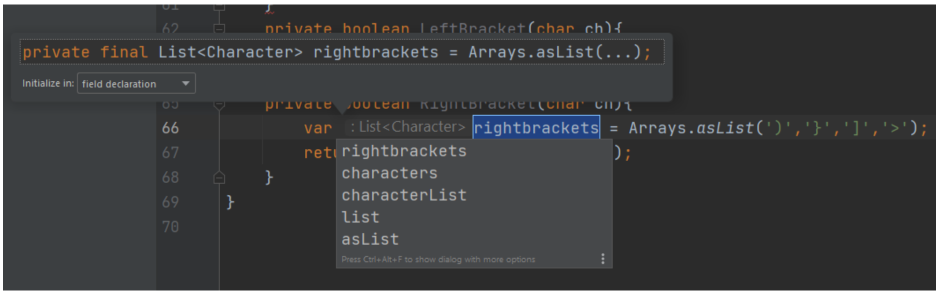
Step 1:- Take cursor to the field that I am going to extract or select the field . In the picture below I want rightbrackets field form RightBracket method to be extracted. So I select "rightbrackets".



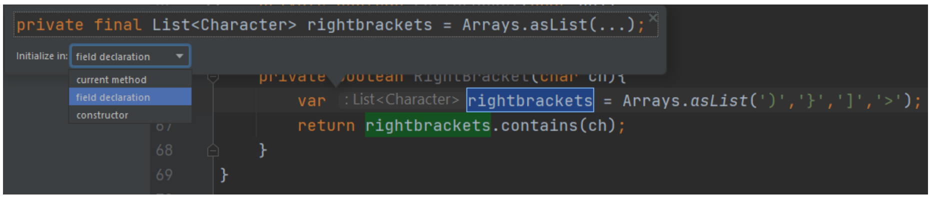
Step 2:- Right Click on the mouse then Reactor→ Introduce Field.



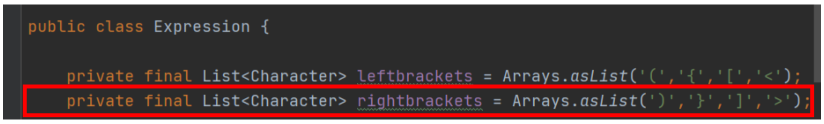
Step 3: Then we got something like the image below-



Step 4: Form initialize in dropdown menu select field declaration and hit enter



Step 5: We got our rightbrackets filed extracted at the top of the class like below:-



After Completing All the steps and extract the fields we can reuse the two arraylist as much time as we want memory is save and time also. Otherwise, as many times isBalanced method is called the ArrayList would have been initialized multiple times.

So, Now Mosh Solution 3(More Readable Refactored+):-

```
public boolean isBalanced3(String input2){
    Stack<Character> stack2 = new Stack<>();
    for(char ch : input2.toCharArray()){//pura string charcter array te niye iterate
        if(LeftBracket(ch))// left bracket (,{,[,< paile push
            stack2.push(ch);
        if(RightBracket(ch)) {// right bracket ),},],> paile pop kore compare
            if(stack2.empty()) return false;

            var top = stack2.pop();
            if(!BracketMatched(top,ch)) return false;
        }
    }
    return stack2.empty();

}
private boolean LeftBracket(char ch){
    return leftbrackets.contains(ch);
}
private boolean RightBracket(char ch){
    return rightbrackets.contains(ch);
}
private boolean BracketMatched(char left,char right){
    return  leftbrackets.indexOf(left)==rightbrackets.indexOf(right);// index same hole match korse
}
```

(V-(11+12))- Now We will be built stack using array

```
import java.util.Arrays;

public class Stack {
    // Basically we will be going to implement the stacks using array
    // where below methods should be there
    //push
    //pop
    //peek
    //isEmpty
    private int count;
    int [] items = new int[5];// stack size limit 5
    public void push(int item){
        if (count==items.length)
            throw new StackOverflowError();

        items[count++]=item;
    }
    public int peek(){
        if(count==0)
            throw new IllegalStateException();
        return items[count-1];// cause top er value count er cheye 1 kom
        //kenona item add hoye count er value 1 bere jabe but last item jeta
        // stack e add hoise seta toh count - 1 ei thakbe
    }
    public int pop(){
        if(count==0)
            throw new IllegalStateException();

        return items[--count];// cause top er value count er cheye 1 kom
    }
    public boolean isEmpty(){
        return count==0;
    }
    @Override
    public String toString(){
        var contents = Arrays.copyOfRange(items,0,count);// eikhane count porjonto
joto gula items ase seigula copy holo
       return Arrays.toString(contents);
    }
}
```

Mosh Stack Exercises-(V-13):- Seems Very Easy But actually implementation is tricky.

Stacks

Exercises

1- Implement two stacks in one array. Support these operations:

```
push1() // to push in the first stack
push2() // to push in the second stack
pop1()
pop2()
isEmpty1()
isEmpty2()
isFull1()
isFull2()
```

Make sure your implementation is space efficient. (hint: do not allocate the same amount of space by dividing the array in half.)

Solution: TwoStacks

2- Design a stack that supports push, pop and retrieving the minimum value in constant time.

For example, we populate our stack with [5, 2, 10, 1] (from left to right).

```
stack.min() // 1
stack.pop()
stack.min() // 2
```

Solution: MinStack

Code of TwoStacks:-

```
import java.util.Arrays;

public class TwoStacks {
    private int top1;
    private int top2;
    private int[] items;

    public TwoStacks(int capacity) {
        if (capacity <= 0)
            throw new IllegalArgumentException("capacity must be 1 or greater.");

        items = new int[capacity];
        top1 = -1;
        top2 = capacity;
    }

    public void push1(int item) {
        if (isFull1())
            throw new IllegalStateException();

        items[++top1] = item;
    }

    public int pop1() {
        if (isEmpty1())
            throw new IllegalStateException();

        return items[top1--];
    }

    public boolean isEmpty1() {
        return top1 == -1;
    }

    public boolean isFull1() {
        return top1 + 1 == top2;
    }

    public void push2(int item) {
        if (isFull2())
            throw new IllegalStateException();

        items[--top2] = item;
    }
```

```
    public int pop2() {
        if (isEmpty2())
            throw new IllegalStateException();

        return items[top2++];
    }

    public boolean isEmpty2() {
        return top2 == items.length;
    }
    public boolean isFull2() {
        return top2 - 1 == top1;
    }

    @Override
    public String toString() {
        return Arrays.toString(items);
    }
}
```

Code of Minstack :-

```
// We need two stacks to implement a min stack.
// One stack holds the values, the other stack
// (called minStack) holds the minimums.
public class MinStack {
    private Stack stack = new Stack();
    private Stack minStack = new Stack();

    public void push(int item) {
        stack.push(item);

        if (minStack.isEmpty())
            minStack.push(item);
        else if (item < minStack.peek())
            minStack.push(item);
    }
```

```
    public int pop() {
        if (stack.isEmpty())
            throw new IllegalStateException();

        var top = stack.pop();

        if (minStack.peek() == top)
            minStack.pop();

        return top;
    }

    public int min() {
        return minStack.peek();
    }
}
```

SUMMARY-(V-14)

STACKS

- Last-In First-Out (LIFO) **[Undo or doing operations in reverse order]**
- Can be implemented using Arrays / Linked Lists
- All operations run in O(1)

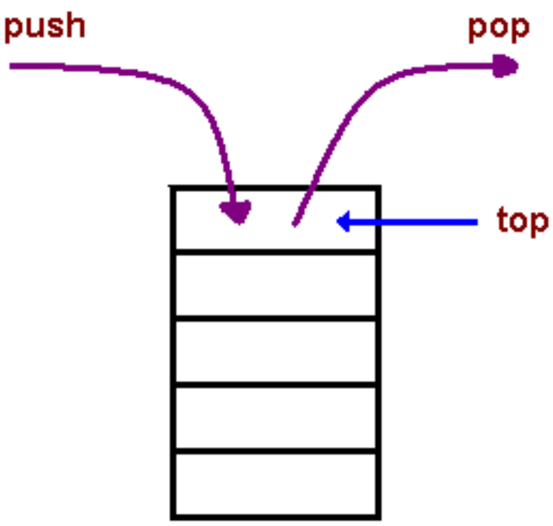
Stacks

Table of contents

Introduction

A Stack is a restricted ordered sequence in which we can only add to and remove from one end — the **top** of the stack. Imagine stacking a set of books on top of each other — you can **push** a new book on **top** of the stack, and you can **pop** the book that is currently on the top of the stack. You are not, strictly speaking, allowed to add to the middle of the stack, nor are you allowed to remove a book from the middle of the stack. The only book that can be taken out of the stack is the **most recently added** one; a stack is thus a "last in, first out" (LIFO) data structure.

We use stacks everyday — from finding our way out of a maze, to evaluating postfix expressions, "undoing" the edits in a word-processor, and to implementing recursion in programming language runtime environments.



Three basic stack operations are:

- **push(obj)**: adds `obj` to the top of the stack ("overflow" error if the stack has fixed capacity, and is full)
- **pop**: removes and returns the item from the top of the stack ("underflow" error if the stack is empty)
- **peek**: returns the item that is on the top of the stack, but does not remove it ("underflow" error if the stack is empty)

The following shows the various operations on a stack.

```
Java statement                     resulting stack
------------------------------------------------
Stack s = new Stack();
                                    ----- (empty stack)



S.push(7);
                                    | 7 |  <-- top
                                    -----

S.push(2);
                                    | 2 |  <-- top
                                    | 7 |
                                    -----

S.push(73);
                                    |73 |  <-- top
                                    | 2 |
                                    | 7 |
                                    -----

S.pop();
                                    | 2 |  <-- top
                                    | 7 |
                                    -----
S.pop();
                                    | 7 |  <-- top
                                    -----
S.pop();
                                    -----  (empty stack)

S.pop();
                                    ERROR "stack underflow"
```

Back to Table of contents

Stack applications

- **Reverse**: The simplest application of a stack is to reverse a word. You push a given word to stack – letter by letter – and then pop letters from the stack. Here's the trivial algorithm to print a word in reverse:

```
begin with an empty stack and an input stream.
while there is more characters to read, do:
   read the next input character;
   push it onto the stack;
end while;
while the stack is not empty, do:
   c = pop the stack;
   print c;
end while;
```

- **Undo**: Another application is an "undo" mechanism in text editors; this operation is accomplished by keeping all text changes in a stack. Popping the stack is equivalent to "undoing" the last action. A very similar one is going back pages in a browser using the *back* button; this is accomplished by keeping the pages visited in a stack. Clicking on the *back* button is equivalent to going back to the most-recently visited page prior to the current one.
- **Expression evaluation**: When an arithmetic expression is presented in the *postfix* form, you can use a stack to evaluate it to get the final value. For example: the expression `3 + 5 * 9` (which is in the usual *infix* form) can be written as `3 5 9 * +` in the *postfix*. More interestingly, postfix form removes all parentheses and thus all implicit precedence rules; for example, the infix expression `((3 + 2) * 4) / (5 - 1)` is written as the postfix `3 2 + 4 * 5 1 - /`. You can now design a calculator for expressions in postfix form using a stack.

  The algorithm may be written as the following:

```
begin with an empty stack and an input stream (for the expression).
while there is more input to read, do:
   read the next input symbol;
   if it's an operand,
      then push it onto the stack;
   if it's an operator
      then pop two operands from the stack;
           perform the operation on the operands;
           push the result;
end while;
// the answer of the expression is waiting for you in the stack:
pop the answer;
```

  Let's apply this algorithm to to evaluate the postfix expression `3 2 + 4 * 5 1 - /` using a stack.

```
 Stack    Expression

|   |
 ---        3 2 + 4 * 5 1 - /
(empty)

| 3 |       2 + 4 * 5 1 - /
 ---

| 2 |
| 3 |       + 4 * 5 1 - /
 ---

| 5 |       4 * 5 1 - /
 ---

| 4 |
| 5 |       * 5 1 - /
 ---

| 20|       5 1 - /
 ---

| 5 |
| 20|       1 - /
 ---
```

```
| 1 |
| 5 |
| 20|       - /
 ---

| 4 |
| 20|       /
 ---

| 5 |      (finished. the result is on top of the stack)
 ---
```

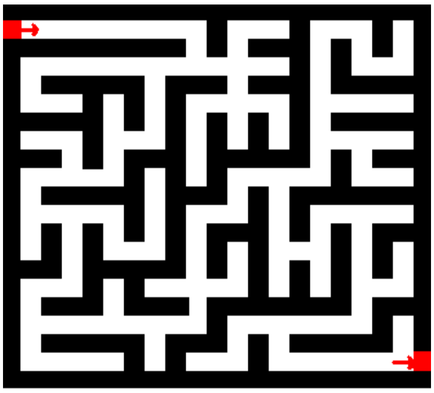
- **Parentheses matching**: We often have a expressions involving "()[]{}" that requires that the different types parentheses are *balanced*. For example, the following are properly balanced:
    - `(a (b + c) + d)`
    - `[ (a b) (c d) ]`
    - `( [a {x y} b] )`

  But the following are not:
    - `(a (b + c) + d`
    - `[ (a b] (c d) )`
    - `( [a {x y) b] )`

  The algorithm may be written as the following:

```
begin with an empty stack and an input stream (for the expression).
while there is more input to read, do:
   read the next input character;
   if it's an opening parenthesis/brace/bracket ("(" or "{" or "[")
      then push it onto the stack;
   if it's a closing parenthesis/brace/bracket (")" or "}" or "]")
      then pop the opening symbol from stack;
           compare the closing with opening symbol;
           if it matches
              then continue with next input character;
           if it does not match
              then return false;
end while;
// all matched, so return true
return true;
```

- **Backtracking**: This is a process when you need to access the most recent data element in a series of elements. Think of a labyrinth or maze - how do you find a way from an entrance to an exit?



Once you reach a dead end, you must backtrack. But backtrack to where? to the previous choice point. Therefore, at each choice point you store on a stack all possible choices. Then backtracking simply means popping a next choice from the stack.

- **Language processing**:
  - space for parameters and local variables is created internally using a stack (*activation records*).
  - compiler's syntax check for matching braces is implemented by using stack.
  - support for recursion

Back to Table of contents

Stack implementation

In the standard library of classes, the data type stack is an *adapter* class, meaning that a stack is built on top of other data structures. The underlying structure for a stack could be an array, a vector, an ArrayList, a linked list, or any other sequence (`Collection` class in JDK). Regardless of the type of the underlying data structure, a `Stack` must implement the same functionality. This is achieved by providing an interface:

```
public interface Stack {
    // The number of items on the stack
    int size();
    // Returns true if the stack is empty
    boolean isEmpty();
    // Pushes the new item on the stack, throwing the
    // StackOverflowException if the stack is at maximum capacity. It
    // does not throw an exception for an "unbounded" stack, which
    // dynamically adjusts capacity as needed.
    void push(Object o) throws StackOverflowException;
    // Pops the item on the top of the stack, throwing the
    // StackUnderflowException if the stack is empty.
    Object pop() throws StackUnderflowException;
    // Peeks at the item on the top of the stack, throwing
    // StackUnderflowException if the stack is empty.
    Object peek() throws StackUnderflowException;
    // Returns a textual representation of items on the stack, in the
    // format "[ x y z ]", where x and z are items on top and bottom
    // of the stack respectively.
    String toString();
    // Returns an array with items on the stack, with the item on top
    // of the stack in the first slot, and bottom in the last slot.
```

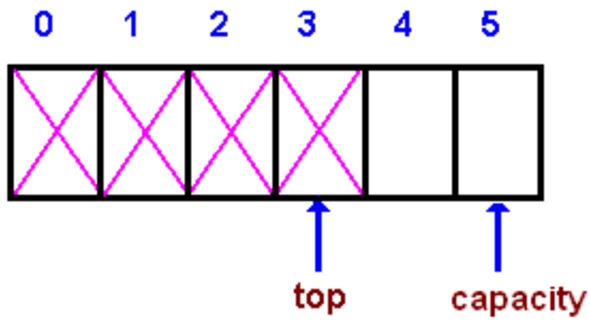
```
    Object[] toArray();
    // Searches for the given item on the stack, returning the
    // offset from top of the stack if item is found, or -1 otherwise.
    int search(Object o);
}
```

One requirement of a `Stack` implementation is that the `push` and `pop` operations run in *constant time*, that is, the time taken for stack operation is independent of how big or small the stack is.

Back to Table of contents

Array based implementation

In an array-based implementation we add the new item being pushed at the end of the array, and consequently pop the item from the end of the array as well. This way, there is no need to shift the array elements. The top of the stack is always the last used slot in the array, so we can use `size-1` to refer to the top of the stack element instead of having to keep a separate field. The *top* of the stack is not defined for an empty stack.



In a *bounded* or fixed-size stack abstraction, the capacity stays unchanged, therefore when *top* reaches *capacity*, the stack object throws an exception.

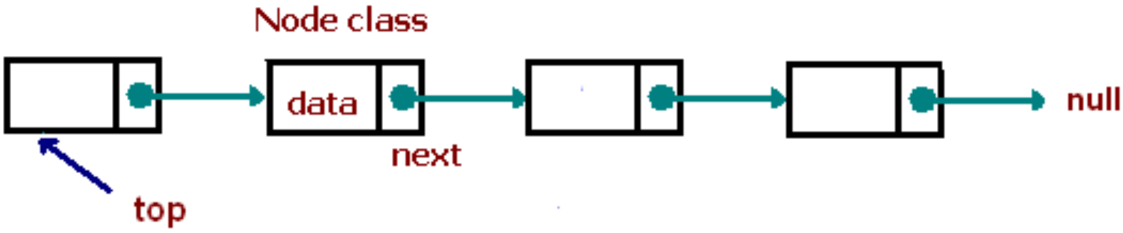
In an *unbounded* or dynamic stack abstraction when *top* reaches *capacity*, we double up the stack size. The following shows a partial array-based implementation of an *unbounded* stack.

```
public class ArrayStack implements Stack {
    private Object[] data;      // The array container
    private int      size;      // The number of items in the stack

    // Default initial capacity, which may then dynamically change
    private static final int DEF_INIT_CAPACITY = 100;

    public ArrayStack() {
        data = new Object[DEF_INIT_CAPACITY];
        size = 0;
    }
}
```

Back to Table of contents

Linked list based implementation

Linked List-based implementation provides the best (from the efficiency point of view) dynamic stack implementation.



The following shows a partial head-referenced singly-linked based implementation of an *unbounded* stack. In an singly-linked list-based implementation we add the new item being pushed at the beginning of the array (why?), and consequently pop the item from the beginning of the list as well.

```
public class ListStack implements Stack {
    private Node head;          // Reference to the top of the stack
    private int  size;          // The number of items in the stack

    public ListStack() {
        head = null;
        size = 0;
    }
}
```

Back to [Table of contents]

Subject: Date: Sat Sun Mon Tue Wed Thu Fri

Stack

1. push( ) pass → Insert
2. pop( ) No pass → To ~~get~~ remove out at the last value what I insert
3. peek() → which I can insert ~~first~~ just see
4. isEmpty () → If I remove anything then it say false
5. stack overflow Exception

push (10)

(20)

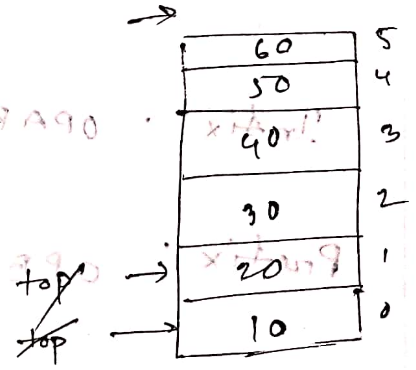
(30)

(40)

(50)

(60)

(90) → stack overflow Exception



pop (50)

(40)

(30)

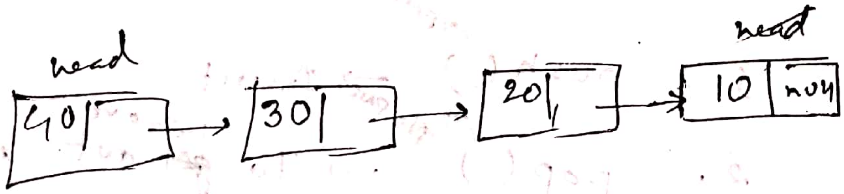
(20)

(10)

(0) → stack underflow Exception.

head = null

push (20)
(30)
(40)



In lisklist there will be no space of stack overflow Exception.

Infix : OPARAND1 OPERATOR OPARAND 2

Prefix : OPERATOR OPARAND1 OPARAD2

Postfix : OPERAND1 OPERAND 2 OPARATOR

* OPERATOR Peferances

( )

{ }

[ ]

x ^ / %

+ -

For same priority left to right.

(i) Infix : A−B/(C*D^E)

(C*DE^)

CDE^*

BCDE^*/

Postfix : ABCDE^*/−

Prefix : C*^DE

B/*C^DE

A− /B*C^DE

−A/B*C^DE

(ii) Infix : 2*3/(2−1)+5*3

Postfix : 2*3/21−+5*3

2*321/−+53*

~~2321*/−53−~~

~~23*21/−~~

23*/21−+53*

23*21−/53*+

23*21−/53*+

Prefix : 2*3/−21+5*3

*23/−21+*53

~~/*23−21+21~~ −

+/*23−21*53

(iii) Infix : (A+B/C * (D+E) - F)

Postfix : (A+B/C * DE+ - F)

(A + BC/ * DE + - F)

A + BC/·DE * + - F

ABC/DE + * + F -

Infix to Postfix using stack Rule's

1st rule : ~~Know to~~ No ~~same~~ oparator's of the same priority can stay together.

The oparator that was already in the stack will be poped out.

2nd rule : ~~Know~~ No lower priority oparetor can stay on top of a higher priority oparetor.

The higher priority oparetor will be poped out.

3r rule : Wherever a closing bracket is found all the opatores in between the braces/bracket will be poped out from Right to left.

Subject:

Date:

Sat □ Sun □ Mon □ Tue □ Wed □ Thu □ Fri

<u>9th rule</u> : At the end, if any ~~opatee~~ ~~store~~ left in the stack there will be poped from right to left.

eq : (A+B/C * (D+E) - F)

Expression	stack (operator)	Operand + popped valus
(	(	
A		A
+	(+	
B		AB
/	(+/	
C		ABC
*	(+~~/~~*	ABC/
(	(+*(	
D		ABC/D
+	(+*(+	
E		ABC/DE
)	(+*~~(+)~~ poped	ABC/DE+
-	(~~*~~~~+~~- pop pop	ABC/DE+*+
F	~~(~~ (-)	ABC/DE+*+F
)		ABC/DE+*+F-

Date:

Sat Sun Mon Tue Wed Thu Fri

Subject:

( ) { } [ ]

1. * / %

2. + -

3. < <= > >= priority decresing

4. == !=

5. &&

6. ||

[(4 * 2 > 9/3 + 5 && 4 = = 6 - 2) ! = 5 < 5] = = 6 > 7 % 1

Expression	stack (operator)	Opparand + popped value
[	[	4
(	[(	42
4	[(*	42*
*	[(*> poped	42*9
2	[(>	42*93
>	[(>\	42*93\
9	E(>\	42*93\5
/	[(>\+ poped	42*93\5+>
3	[(>+	
+	[(>+&& po poped	
5	[(&&	
&&		

Subject: Date: Sat □Sun□Mon□Tue□Wed□Thu□Fri

4	[(&&	42*93\5+>4
==	[(&&==	42*93\5+>462
6	[(&&==-	42*93\5+>4 62
-	[(&&==-	42*93\5+>462-== &&
2	[(&&==-)	
)		**Final answer** 42*93\5+>462-== &&55<!=671%>==
!=	[!=	42*93\5+>462-==&&
5	[!=	42*93\5+>462-==&&5
<	[!=<	42*93\5+>462-==&&55
5	[!=<]	42*93\5+>462-==&&55<!=
]	==	42*93\5+>462-==&&55<!=6
==	==>	
6		42*93\5+>462-==&&55<!=67
>	==>%	
7		42*93\5+>462-==&&55<!=671
%		42*93\5+>462-=&&55
1		<!=671%>==

Papers King