ArrayList

#### **ArrayList**

ArrayList is a part of collection framework and is present in java.util package.  It provides a dynamic way of manipulating data.  Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed.  Some of the notable characteristics of ArrayList are the following:

* ArrayList inherits AbstractList class and implements List interface.
* ArrayList is initialized by size, however, the size can increase if collection grows or shrunk if objects are removed from the collection.
* Java ArrayList allows us to randomly access the list.
* Use a wrapper class if an ArrayList can not use primitive types, like int, char, etc.
* ArrayList in Java can be seen as similar to a vector in C++.

#### **ArrayList Constructors**

1. **ArrayList()**: This constructor is used to build an empty array list
2. **ArrayList(Collection c)**: This constructor is used to build an array list initialized with the elements from collection c
3. **ArrayList(int capacity)**: This constructor is used to build an array list with initial capacity being specified

#### **Basic Structure of an ArrayList**

ArrayList<Integer> arrayList1 = new ArrayList<Integer>();  
  
ArrayList<String> listarrayList2=new ArrayList<String>();  
  
ArrayList<Boolean> arrayList3 = new ArrayList<Boolean>();

#### **ArrayList Methods**

|  |  |
| --- | --- |
| add(int index, Object element) | This method is used to insert a specific element at a specific position index in a list. |
| add(Object o) | This method is used to append a specific element to the end of a list. |
| addAll(Collection C) | This method is used to append all the elements from a specific collection to the end of the mentioned list, in such an order that the values are returned by the specified collection’s iterator. |
| addAll(int index, Collection C) | Used to insert all of the elements starting at the specified position from a specific collection into the mentioned list. |
| clear() | This method is used to remove all the elements from any list. |
| clone() | This method is used to return a shallow copy of an ArrayList. |
| contains? (Object o) | Returns true if this list contains the specified element. |
| ensureCapacity?(int minCapacity) | Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument. |
| forEach?(Consumer<? super E> action) | Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception. |
| get?(int index) | Returns the element at the specified position in this list. |
| indexOf(Object O) | The index the first occurrence of a specific element is either returned or -1 in case the element is not in the list. |
| isEmpty?() | Returns true if this list contains no elements. |
| lastIndexOf(Object O) | The index of the last occurrence of a specific element is either returned or -1 in case the element is not in the list. |
| listIterator?() | Returns a list iterator over the elements in this list (in proper sequence). |
| listIterator?(int index) | Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list. |
| remove?(int index) | Removes the element at the specified position in this list. |
| remove?(Object o) | Removes the first occurrence of the specified element from this list, if it is present. |
| removeAll?(Collection c) | Removes from this list all of its elements that are contained in the specified collection. |
| removeIf?(Predicate filter) | Removes all of the elements of this collection that satisfy the given predicate. |
| removeRange?(int fromIndex, int toIndex) | Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive. |
| retainAll?(Collection<?> c) | Retains only the elements in this list that are contained in the specified collection. |
| set?(int index, E element) | Replaces the element at the specified position in this list with the specified element. |
| size?() | Returns the number of elements in this list. |
| spliterator?() | Creates a late-binding and fail-fast Spliterator over the elements in this list. |
| subList?(int fromIndex, int toIndex) | Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive. |
| toArray() | This method is used to return an array containing all of the elements in the list in the correct order. |
| toArray(Object[] O) | It is also used to return an array containing all of the elements in this list in the correct order same as the previous method. |
| trimToSize() | This method is used to trim the capacity of the instance of the ArrayList to the list’s current size. |

# Stack

**A stack** is a way to group things together by placing one thing on top of another and then removing things one at a time from the top of the stack.

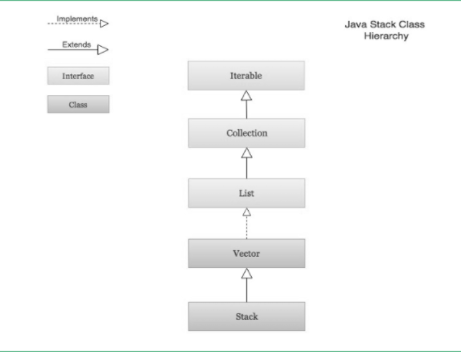
In computer science, a **stack** is **a Last-In, First-Out (L I F O)** abstract data type and data structure. A stack can have any abstract data type as an element but is characterized by only two fundamental operations: push and pop.

The **push**operation adds to the top of the list, hiding any items already on the stack, or initializing the stack if it is empty. The **pop**operation removes an item from the top of the list and returns this value to the caller. A pop either reveals previously concealed items or results in an empty list.

#### **Stack as Data Structure**

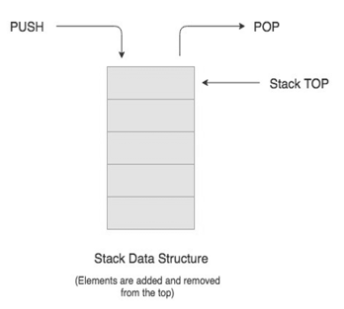
A **stack**is a restricted data structure because only a small number of operations are performed on it. The nature of the pop and push operations also means that stack elements have a natural order. Elements are removed from the stack in the reverse order to the order of their addition: therefore, the lower elements are typically those that have been in the list the longest.

A stack is a linear data structure that follows a particular order in which the operations are performed.  It is a part of Java’s collections framework. The following figure shows the class hierarchy of the Stack.



The Stack class extends Vector which implements the List interface.  A Vector is a re-sizable collection.  It grows its size to accommodate new elements and shrinks the size when the elements are removed.  Since the Stack class extends Vector; it also grows and shrinks its size as needed when new elements are added or removed.

#### **Visualization of a Stack**



#### **Real-life Example of a Stack**

Consider an example of plates stacked over one another in the canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO(Last In First Out)/FILO(First In Last Out) order.

#### **Creation of Stack**

Stack class is used by importing the **java.util library**.  A Stack object is created using the following structure.

Stack<BaseType> s = new Stack<BaseType>();

Just like ArrayList, Stack requires a base type to be an object

#### **Stack Operations**

The **push( )** operation is used both to initialize the stack and to store values to it. It is responsible for inserting (copying) the value into the array and for incrementing the element counter (size). In a responsible implementation, it is also necessary to check whether the array is already full to prevent an overrun.

It is an operation used to insert or add a data item or element to the stack. The push ( ) method requires an item when called.  The push operation is used for insertion of new elements in the Stack.

The **pop( )** operation is responsible for removing a value from the stack, and decrementing the value of size. A responsible implementation will also need to check that the array is not already empty.

It is an operation used to delete or remove a data item or element at the top of the stack. The pop ( ) method returns the item being removed in the Stack.  The pop operation is used for the deletion of the top element of the Stack

The **size( )**operation is an operation to determine the size (number of items) of the Stack. It is used mainly to control loops. The size operation is used for checking the size of a Stack.

The **peek( )** operation is a method that looks at the item at the top of a stack. The peek ( ) method returns the item at the top without removing it.  The peek operation is used to determine what item is at the top of the Stack

The **search ( )** member method is a method that returns the position (in number) of an item from the top of a stack. The search ( ) method requires the desired item when called.  The search operation is used to determine the position of the item.

The **empty ( )** member method is a method that tests if a stack object is empty or not. The empty ( ) method returns either a boolean value of true or false. The method returns true if the stack is empty and false if it still contains an item or element.  The empty operation is used for checking if the Stack is empty.

#### **Infix to Postfix Conversion Using Stack**

One of the applications of Stack is in the conversion of arithmetic expressions in high-level programming languages into machine-readable form.  As our computer system can only understand and work on a binary language, it assumes that an arithmetic operation can take place in two operands only e.g., **A+B, C\*D, D/A,** etc.

But in our usual form, an arithmetic expression may consist of more than one operator and two operands e.g. **(A+B)\*C(D/(J+D))**.

These complex arithmetic operations can be converted into polish notation using stacks which then can be executed in two operands and an operator form.

#### **Infix Expression**

Infix expression follows the scheme of **<operand><operator><operand>** i.e. an <operator> is preceded and succeeded by an <operand>.  The expression of the form **a op b**.  When an operator is in-between every pair of operands.  Such an expression is termed infix expression, e.g., **A+B**.

#### **Postfix Expression**

Postfix expression follows the scheme of **<operand><operand><operator>** i.e. an <operator> is succeeded by both the <operand>.  The expression of the form **a b op**. When an operator is followed for every pair of an operand, e.g., **AB+**.

#### **Why Postfix Representation of the Expression is Important?**

The compiler scans the expression either from left to right or from right to left.  Consider the below expression:

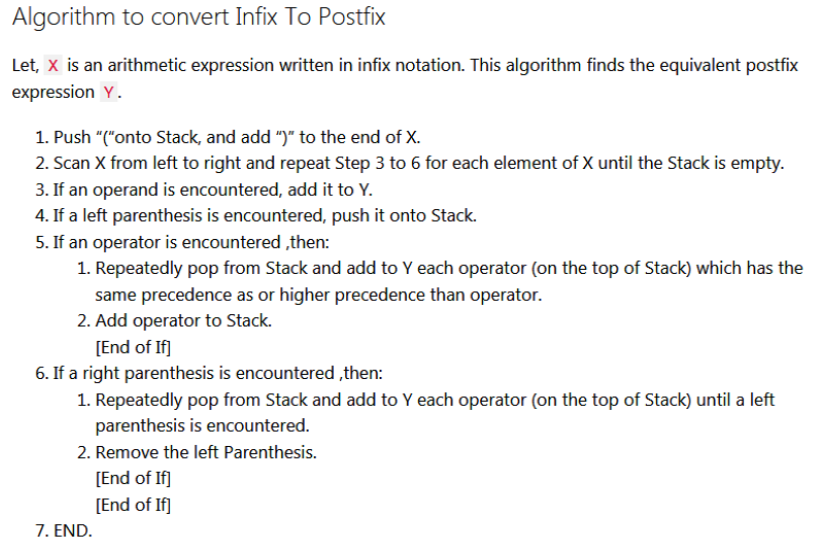
a op1 b op2 c op3 d  
  
If op1 = +, op2 = \*, op3 = +

The compiler first scans the expression to evaluate the expression**b \* c**, then again scan the expression to add a to it. The result is then added to d after another scan.  The repeated scanning makes it very in-efficient. It is better to convert the expression to postfix(or prefix) form before evaluation.  The corresponding expression in the postfix form is **abc\*+d+**. The postfix expressions can be evaluated easily using a stack.

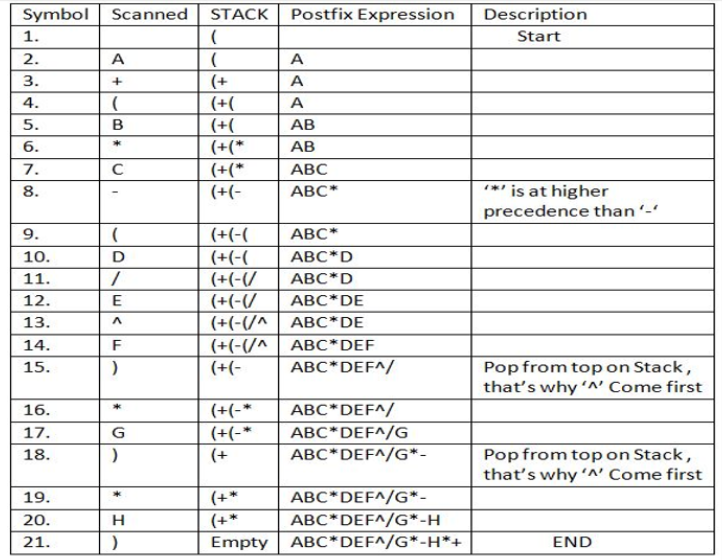
**Algorithm to convert Infix To Postfix**

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,
   * If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a ‘(‘ ), push it.
   * Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
4. If the scanned character is an ‘(‘, push it to the stack.
5. If the scanned character is an ‘)’, pop the stack and output it until a ‘(‘ is encountered, and discard both the parenthesis.
6. Repeat steps 2-6 until infix expression is scanned.
7. Print the output
8. Pop and output from the stack until it is not empty.

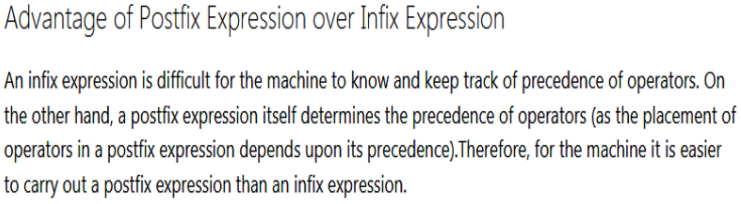
#### **Algorithm to convert Infix To Postfix**



#### **Conversion of Infix to Postfix**



Advantage of postfix expression over infix expression



**Prefix to Infix Conversion**

* **Infix**: An expression is called the Infix expression if the operator appears in between the operands in the expression. Simply of the form (operand1 operator operand2).

Example : (A+B) \* (C-D)

* **Prefix**: An expression is called the prefix expression if the operator appears in the expression before the operands. Simply of the form (operator operand1 operand2).

Example : \*+AB-CD (Infix : (A+B) \* (C-D) )

Given a Prefix expression, convert it into an Infix expression.  Computers usually do the computation in either prefix or postfix (usually postfix).  But for humans, its easier to understand an Infix expression rather than a prefix.  Hence conversion is needed for human understanding.

