# Stack Class in Java

geeksforgeeks.org/stack-class-in-java/

February 4, 2016

Suggest changes

Like Article

Like

Save

Share

Report

Follow

Java Collection framework provides a Stack class that models and implements a **Stack data structure**. The class is based on the basic principle of **LIFO**(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 `Stack` class extends `Vector` and provides additional functionality specifically for stack operations, such as **`push`**, **`pop`**, **`peek`**, **`empty`**, and **`search`**.
- The `Stack` class can indeed be referred to as a subclass of `Vector`, inheriting its methods and properties.

**Example:**

Java

```
// Java Program Implementing Stack Class
import java.util.Stack;

public class StackExample
{
    public static void main(String[] args)
    {
        // Create a new stack
        Stack<Integer> s = new Stack<>();

        // Push elements onto the stack
        s.push(1);
        s.push(2);
        s.push(3);
        s.push(4);

        // Pop elements from the stack
        while(!s.isEmpty()) {
            System.out.println(s.pop());
        }
    }
}
```

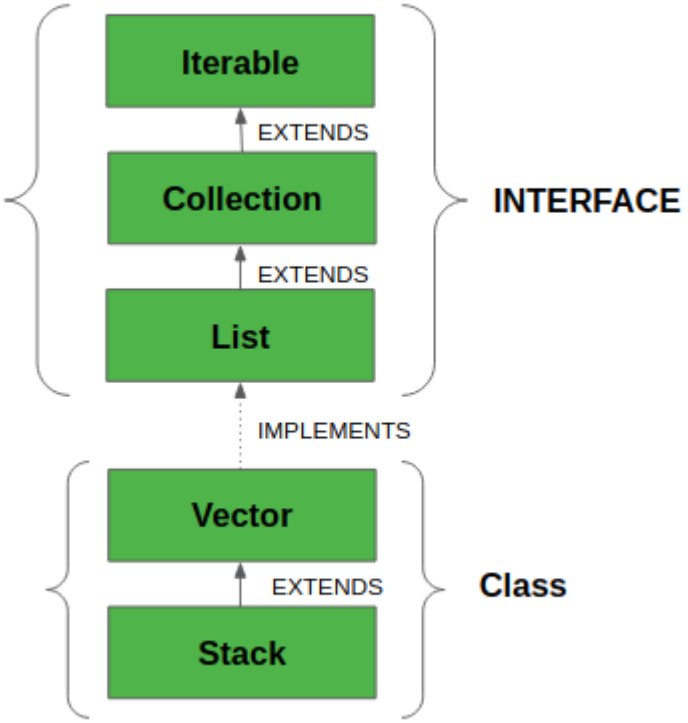
### Output

```
4
3
2
1
```

### Explanation:

- In this example, we first import the Stack class from the java.util package.
- We then create a new Stack object called stack using the default constructor.
- We push four integers onto the stack using the push() method.
- We then pop the elements from the stack using the pop() method inside a while loop.
- The isEmpty() method is used to check if the stack is empty before attempting to pop an element.
- This code creates a stack of integers and pushes 4 integers onto the stack in the order 1 -> 2 -> 3 -> 4.
- We then pop elements from the stack one by one using the pop() method, which removes and returns the top element of the stack.
- Since the stack follows a last-in-first-out (LIFO) order, the elements are popped in the reverse order of insertion, resulting in the output shown above.

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




The class supports one *default constructor* **Stack()** which is used to *create an empty stack*.

## Declaration of Stack

> public class Stack<E> extends Vector<E>

**All Implemented Interfaces:**

- **Serializable:** It is a marker interface that classes must implement if they are to be serialized and deserialized.
- **Cloneable:** This is an interface in Java which needs to be implemented by a class to allow its objects to be cloned.
- **Iterable<E>:** This interface represents a collection of objects which is iterable — meaning which can be iterated.
- **Collection<E>:** A Collection represents a group of objects known as its elements. The Collection interface is used to pass around collections of objects where maximum generality is desired.
- **List<E>:** The List interface provides a way to store the ordered collection. It is a child interface of Collection.
- **RandomAccess:** This is a marker interface used by List implementations to indicate that they support fast (generally constant time) random access.

## How to Create a Stack?

In order to create a stack, we must import **java.util.stack** package and use the Stack() constructor of this class. The below example creates an empty Stack.

```
Stack<E> stack = new Stack<E>();
```

Here E is the type of Object.

**Example:**

Java

```
// Java code for stack implementation
import java.util.Stack;

class Main
{
    // Pushing element on the top of the stack
    static void stack_push(Stack<Integer> stack)
    {
        for(int i = 0; i < 5; i++)
        {
            stack.push(i);
        }
    }

    // Popping element from the top of the stack
    static void stack_pop(Stack<Integer> stack)
    {
        System.out.println("Pop Operation:");

        for(int i = 0; i < 5; i++)
        {
            Integer y = (Integer) stack.pop();
            System.out.println(y);
        }
    }

    // Displaying element on the top of the stack
    static void stack_peek(Stack<Integer> stack)
    {
        Integer element = (Integer) stack.peek();
        System.out.println("Element on stack top: " + element);
    }

    // Searching element in the stack
    static void stack_search(Stack<Integer> stack, int element)
    {
        Integer pos = (Integer) stack.search(element);

        if(pos == -1)
            System.out.println("Element not found");
        else
            System.out.println("Element is found at position: " + pos);
    }


    public static void main (String[] args)
    {
        Stack<Integer> stack = new Stack<Integer>();

        stack_push(stack);
        stack_pop(stack);
        stack_push(stack);
        stack_peek(stack);
        stack_search(stack, 2);
        stack_search(stack, 6);
    }
}
```

## Output

```
Pop Operation:
4
3
2
1
0
Element on stack top: 4
Element is found at position: 3
Element not found
```

## Performing various operations on Stack class

**1. Adding Elements:** In order to add an element to the stack, we can use the *push() method*. This **push()** operation place the element at the top of the stack.

Java

```
// Java program to add the
// elements in the stack
import java.io.*;
import java.util.*;

class StackDemo {

      // Main Method
    public static void main(String[] args)
    {

        // Default initialization of Stack
        Stack stack1 = new Stack();

        // Initialization of Stack
        // using Generics
        Stack<String> stack2 = new Stack<String>();

        // pushing the elements
        stack1.push("4");
        stack1.push("All");
        stack1.push("Geeks");

        stack2.push("Geeks");
        stack2.push("For");
        stack2.push("Geeks");

          // Printing the Stack Elements
        System.out.println(stack1);
        System.out.println(stack2);
    }
}
```

### Output

```
[4, All, Geeks]
[Geeks, For, Geeks]
```

**2. Accessing the Element:** To retrieve or fetch the first element of the Stack or the element present at the top of the Stack, we can use **peek()** method. The element retrieved does not get deleted or removed from the Stack.

Java

```
// Java program to demonstrate the accessing
// of the elements from the stack
import java.util.*;
import java.io.*;

public class StackDemo {

      // Main Method
    public static void main(String args[])
    {
        // Creating an empty Stack
        Stack<String> stack = new Stack<String>();

        // Use push() to add elements into the Stack
        stack.push("Welcome");
        stack.push("To");
        stack.push("Geeks");
        stack.push("For");
        stack.push("Geeks");

        // Displaying the Stack
        System.out.println("Initial Stack: " + stack);

        // Fetching the element at the head of the Stack
        System.out.println("The element at the top of the"
                           + " stack is: " + stack.peek());

        // Displaying the Stack after the Operation
        System.out.println("Final Stack: " + stack);
    }
}
```

### Output

```
Initial Stack: [Welcome, To, Geeks, For, Geeks]
The element at the top of the stack is: Geeks
Final Stack: [Welcome, To, Geeks, For, Geeks]
```

**3. Removing Elements:** To pop an element from the stack, we can use the **pop()** method. The element is popped from the top of the stack and is removed from the same.

Java

```
// Java program to demonstrate the removing
// of the elements from the stack
import java.util.*;
import java.io.*;

public class StackDemo {
    public static void main(String args[])
    {
        // Creating an empty Stack
        Stack<Integer> stack = new Stack<Integer>();

        // Use add() method to add elements
        stack.push(10);
        stack.push(15);
        stack.push(30);
        stack.push(20);
        stack.push(5);

        // Displaying the Stack
        System.out.println("Initial Stack: " + stack);

        // Removing elements using pop() method
        System.out.println("Popped element: "
                           + stack.pop());
        System.out.println("Popped element: "
                           + stack.pop());

        // Displaying the Stack after pop operation
        System.out.println("Stack after pop operation "
                           + stack);

       System.out.println("Is stack empty? " + stack.empty()); // Should print false
       // Pop remaining elements
        stack.pop();
        stack.pop();
        stack.pop();

        // Check if the stack is empty
        System.out.println("Is stack empty? " + stack.empty()); // Should print true
    }
}
```

### Output

```
Initial Stack: [10, 15, 30, 20, 5]
Popped element: 5
Popped element: 20
Stack after pop operation [10, 15, 30]
```

The Stack class provides several other methods for manipulating the stack, such as peek() to retrieve the top element without removing it, search() to search for an element in the stack and return its position, and size() to return the current size of the stack. The `Stack` class in Java provides only a default constructor, which creates an empty stack. If you want to create a stack with a specified initial capacity, you would need to extend the `Vector` class (which `Stack` itself extends) and set the initial capacity in the constructor of your custom class.

## Methods in Stack Class

| Method | Description |
| --- | --- |
| empty() | It returns true if nothing is on the top of the stack. Else, returns false. |
| peek() | Returns the element on the top of the stack, but does not remove it. |
| pop() | Removes and returns the top element of the stack. An 'EmptyStackException'<br><br>An exception is thrown if we call pop() when the invoking stack is empty. |
| push(Object element) | Pushes an element on the top of the stack. |
| search(Object element) | It determines whether an object exists in the stack. If the element is found,<br><br>It returns the position of the element from the top of the stack. Else, it returns -1. |

## Methods inherited from class java.util.Vector

| Method | Description |
| --- | --- |
| add(Object obj) | Appends the specified element to the end of this Vector. |
| add(int index, Object obj) | Inserts the specified element at the specified position in this Vector. |
| addAll(Collection c) | Appends all of the elements in the specified Collection to the end of this Vector,<br><br>in the order that they are returned by the specified Collection's Iterator. |
| addAll(int index, Collection c) | Inserts all the elements in the specified Collection into this Vector at the specified position. |
| addElement(Object o) | Adds the specified component to the end of this vector, increasing its size by one. |
| capacity() | Returns the current capacity of this vector. |

| Method | Description |
|---|---|
| clear() | Removes all the elements from this Vector. |
| clone() | Returns a clone of this vector. |
| contains(Object o) | Returns true if this vector contains the specified element. |
| containsAll(Collection c) | Returns true if this Vector contains all the elements in the specified Collection. |
| copyInto(Object []array) | Copies the components of this vector into the specified array. |
| elementAt(int index) | Returns the component at the specified index. |
| elements() | Returns an enumeration of the components of this vector. |
| ensureCapacity(int minCapacity) | Increases the capacity of this vector, if necessary, to ensure that it can hold<br>at least the number of components specified by the minimum capacity argument. |
| equals() | Compares the specified Object with this Vector for equality. |
| firstElement() | Returns the first component (the item at index 0) of this vector. |
| get(int index) | Returns the element at the specified position in this Vector. |
| hashCode() | Returns the hash code value for this Vector. |
| indexOf(Object o) | Returns the index of the first occurrence of the specified element in this vector, or -1<br>if this vector does not contain the element. |
| indexOf(Object o, int index) | Returns the index of the first occurrence of the specified element in this vector, searching forwards from the index, or returns -1 if the element is not found. |
| insertElementAt(Object o, int index) | Inserts the specified object as a component in this vector at the specified index. |
| isEmpty() | Tests if this vector has no components. |
| iterator() | Returns an iterator over the elements in this list in proper sequence. |
| lastElement() | Returns the last component of the vector. |

| Method | Description |
|---|---|
| lastIndexOf(Object o) | Returns the index of the last occurrence of the specified element in this vector, or -1<br><br>If this vector does not contain the element. |
| lastIndexOf(Object o, int index) | Returns the index of the last occurrence of the specified element in this vector,<br><br>searching backward from the index, or returns -1 if the element is not found. |
| 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),<br><br>starting at the specified position in the list. |
| remove(int index) | Removes the element at the specified position in this Vector. |
| remove(Object o) | Removes the first occurrence of the specified element in this Vector If the Vector does not contain the element, it is unchanged. |
| removeAll(Collection c) | Removes from this Vector all of its elements that are contained in the specified Collection. |
| removeAllElements() | Removes all components from this vector and sets its size to zero. |
| removeElement(Object o) | Removes the first (lowest-indexed) occurrence of the argument from this vector. |
| removeElementAt(int index) | Deletes the component at the specified index. |
| removeRange(int fromIndex, int toIndex) | Removes from this list all the elements whose index is between fromIndex, inclusive, and toIndex, exclusive. |
| retainAll(Collection c) | Retains only the elements in this Vector that are contained in the specified Collection. |
| set(int index, Object o) | Replaces the element at the specified position in this Vector with the specified element. |
| setElementAt(Object o, int index) | Sets the component at the specified index of this vector to be the specified object. |
| setSize(int newSize) | Sets the size of this vector. |

| Method | Description |
|---|---|
| size() | Returns the number of components in this vector. |
| subList(int fromIndex, int toIndex) | Returns a view of the portion of this List between fromIndex, inclusive, and toIndex, exclusive. |
| toArray() | Returns an array containing all of the elements in this Vector in the correct order. |
| toArray(Object []array) | Returns an array containing all of the elements in this Vector in the correct order; the runtime<br><br>type of the returned array is that of the specified array. |
| toString() | Returns a string representation of this Vector, containing the String representation of each element. |
| trimToSize() | Trims the capacity of this vector to be the vector's current size. |

## Prioritize use of Deque over Stack

The Stack class in Java is a legacy class and inherits from **Vector in Java**. It is a thread-safe class and hence involves overhead when we do not need thread safety. It is recommended to use **ArrayDeque**for stack implementation as it is more efficient in a single-threaded environment.

Java

```
// A Java Program to show implementation
// of Stack using ArrayDeque

import java.util.*;

class GFG {
    public static void main (String[] args) {
        Deque<Character> stack = new ArrayDeque<Character>();
        stack.push('A');
        stack.push('B');
        System.out.println(stack.peek());
        System.out.println(stack.pop());
    }
}
```

### Output

```
B
B
```

One more reason to use Deque over Stack is Deque has the ability to use streams convert to list with keeping LIFO concept applied while Stack does not.

Java

```
import java.util.*;
import java.util.stream.Collectors;

class GFG {
    public static void main (String[] args) {

          Stack<Integer> stack = new Stack<>();
        Deque<Integer> deque = new ArrayDeque<>();

        stack.push(1);//1 is the top
        deque.push(1);//1 is the top
        stack.push(2);//2 is the top
        deque.push(2);//2 is the top

        List<Integer> list1 = stack.stream().collect(Collectors.toList());//[1,2]
          System.out.println("Using Stack -");
          for(int i = 0; i < list1.size(); i++){
              System.out.print(list1.get(i) + " " );
        }
          System.out.println();

        List<Integer> list2 = deque.stream().collect(Collectors.toList());//[2,1]
          System.out.println("Using Deque -");
          for(int i = 0; i < list2.size(); i++){
              System.out.print(list2.get(i) + " " );
        }
          System.out.println();

    }
}
```

### Output

```
Using Stack -
1 2
Using Deque -
2 1
```